ABSTRACT

This tech report presents formal specifications for the Memoir system and proofs of the system’s correctness. The proofs were constructed manually but have been programatically machine-verified using the TLA+ Proof System. Taken together, the specifications and proofs contain 61 top-level definitions, 182 LET-IN definitions, 74 named theorems, and 5816 discrete proof steps. The proofs address only the safety of the Memoir system, not the liveness of the system. Safety is proven by showing that a formal low-level specification of the Memoir-Basic system implements a formal high-level specification of desired behavior. The proofs then show that a formal specification of the Memoir-Opt system implements the Memoir-Basic system.

TABLE OF CONTENTS

1 INTRODUCTION 2
  1.1 Overview ......................................................... 2
  1.2 Memoir-Basic .................................................. 5
  1.3 Memoir-Opt .................................................... 9
  1.4 Organization .................................................. 11

2 SPECIFICATIONS 18
  2.1 Declarations Common to High- and Low-Level Specs ......................... 19
  2.2 Specification of the High-Level System (Semantics) .......................... 20
  2.3 Primitives Used by the Low-Level Systems .................................... 22
  2.4 Specification of the Memoir-Basic System .................................... 24
  2.5 Specification of the Memoir-Opt System .................................... 31

3 REFINEMENTS AND INVARIANTS 40
  3.1 Refinement 1: Mapping Memoir-Basic State to High-Level State ............... 41
  3.2 Refinement 2: Mapping Memoir-Opt State to Memoir-Basic State ............... 43
  3.3 Invariants Needed to Prove Memoir-Basic Implementation ..................... 47
  3.4 Invariants Needed to Prove Memoir-Basic Invariance .......................... 49

4 PROOFS 51
  4.1 Proof of Type Safety of the High-Level Spec ................................ 52
  4.2 Proof of Lemmas Relating to Types in the Memoir-Basic Spec ................. 54
  4.3 Proof of Type Safety of the Memoir-Basic Spec ................................ 69
  4.4 Proofs of Lemmas that Support Memoir-Basic Invariance Proofs ............... 82
  4.5 Proof of Unforgeability Invariance in Memoir-Basic .......................... 107
  4.6 Proof of Inclusion, Cardinality, and Uniqueness Co-invariance in Memoir-Basic .... 120
  4.7 Proof that Memoir-Basic Spec Implements High-Level Spec .................... 157
  4.8 Proof of Lemmas Relating to Types in the Memoir-Opt Spec ....................... 192
  4.9 Proof of Type Safety of the Memoir-Opt Spec .................................. 213
  4.10 Proof of Lemmas Relating to the Memoir-Opt Refinement ....................... 230
  4.11 Proof of Lemmas Relating to the Memoir-Opt Implementation ................... 271
  4.12 Proof that Memoir-Opt Spec Implements Memoir-Basic Spec ................... 284

ACKNOWLEDGMENTS 390

REFERENCES 390
1. INTRODUCTION

1.1 Overview

This tech report presents formal specifications and safety proofs for the Memoir system. The specifications herein are written in the TLA+ language, and the proofs are written in the TLA+ proof language. Our hope is that a reader unfamiliar with TLA+ can easily understand the prose descriptions in this tech report, along with the textual comments embedded in the specifications and proofs. However, a thorough understanding of this tech report requires a solid knowledge of TLA+. We do not provide even a cursory tutorial of the language herein.

In TLA+, a specification is inductive. The spec describes a set of state variables, the initial values for these variables, and a set of actions that modify the variables. Each action is a relation between a pair of successive states. The temporally earlier state is called the unprimed state, and the temporally later state is called the primed state. (Within the language, the different states are identified by the absence or presence of a prime character following the state variable or expression.)

Our proofs are written in a hierarchical style advocated by Lamport. Each progressive level of the proof contains sub-proofs, each of which proves a single proof step at the prior level. Our proofs contain a total of 5816 discrete proof steps, all in the service of proving 74 named theorems. Although these steps were all written manually, they have been programmatically machine-verified using the TLA+ proof system.

1.1.1 Background—The Memoir System

Memoir is a generic framework for executing modules of code in a protected environment. In particular, Memoir guarantees not only privacy and integrity; it also guarantees state continuity across TCB interruptions. This means that, when a module pauses execution and returns control to the untrusted caller, and then later resumes execution, the module will resume from the same state it was in before it paused.

Understanding the formal specifications in this tech report requires a detailed understanding of the Memoir system, which this tech report does not provide. The reader is referred to our paper describing the Memoir system. The level of knowledge contained in the cited paper will be assumed by the remainder of this tech report.

The associated paper introduces several terms of art with meanings that are specific to Memoir, such as “history”, “history summary”, and “authenticator”. Herein, we supplement these terms with two others that represent intermediate values in the construction of an authenticator:

- **state hash**: a secure hash of a public state and an encrypted private state
- **history state binding**: a secure hash of a history summary and a state hash

Thus, an authenticator is a MAC of a history state binding.

1.1.2 Philosophy and Approach

One common way to formally address the correctness of a system is to state and prove particular properties that the system maintains. Closely related is the approach of proving that the system prevents a particular set of undesirable things from occurring. For instance, in Memoir, we might have asserted that the system is correct if it does not allow a rollback attack to set the service state to a previous state. Although this might seem intuitive, this particular property is in fact too strong, because one could easily define a service that allows transitions to arbitrary previous states. Memoir does not prevent such a service from executing, and there no reason Memoir should be constrained to only run services that disallow repeated entries into the same state.

More importantly, there is a general problem with the approach of stating desirable and/or undesirable properties and proving that they do/don’t follow. The problem is that there is no a priori reason to believe that any particular set of properties sufficiently captures the intended behavior of a system. Even if we were to modify the above non-rollback property to account for systems that allow state re-entry, there is no reason to believe that this is the only important property for the Memoir system to maintain. And, in fact, it is not the only important property: Memoir should also prevent a transition to any state that is not reachable by the service code. Even this is not sufficient, because the service may define multiple states that are independently reachable from the initial state, but that are mutually exclusive in any given execution sequence. We could continue adding and modifying properties, but there is no clear way to know when the set is sufficient to characterize the desired system behavior.

Instead of defining properties, we follow a proof approach encouraged by TLA+. This approach has four main parts, and often includes a fifth. The first part of the approach is to define a high-level specification that describes the intended semantics of the system. The high-level spec is small enough and simple enough that a knowledgeable reader should be able to examine the spec and easily determine whether its semantics are the
right ones. It is, of course, possible to prove properties of the high-level spec, but the hope is that the high-level spec is straightforward enough that its desirability is readily assessable.

The second part of the approach is to define a **low-level specification** that describes the implementation of the system. Whereas a high-level spec typically describes abstract state at the semantic level, a low-level spec typically describes concrete state at the implementation level. Just as it should be easy to determine that the high-level spec describes desirable semantics, it should be easy to determine that the low-level spec accurately characterizes the real hardware and software that implements the system.

The third part of the approach is to define a **refinement** that describes how to interpret any given state of the low-level system as a corresponding state of the high-level system. This is a somewhat subtle concept, and we do not elaborate on it here, although the specific descriptions of refinements below (§1.2.3, §1.3.4) may implicitly provide sufficient edification. A brief and surprisingly entertaining introduction to the topic of refinement is presented in the paper *Refinement in State-Based Formalisms* by Lamport. For the very interested reader, the book *Specifying Systems* describes the concept in depth.

The fourth part of the approach is where the rubber meets the road: a **proof of implementation** that shows that any behavior satisfying the low-level spec, when interpreted according to the refinement, also satisfies the high-level spec. Such a proof may (and ours does) show the mapping from particular actions in the low-level spec to particular actions in the high-level spec. This additional set of cross-spec correspondences provides further understanding of the relationship between the two specs, beyond merely that provided by the state-to-state correspondences established by the refinement.

The fifth (and the only optional) part the approach is to define and prove a set of **inductive invariants** that are maintained by the low-level spec. These inductive invariants may seem similar to the correctness properties we disparaged above; however, they are different in two important respects. First, and most importantly, they are completely in service to the proof of implementation. The only reason the inductive invariants are needed (if they even are) is as a necessary step in the process of proving that the low-level spec satisfies the high-level spec. Consequently, there is no danger that the set will be incomplete in some important way. If the invariants are sufficient to enable the proof of implementation, then the set is complete.

Second, it is not important for these invariants to be understood by a person who merely wants to be confident that the low-level system provides desirable semantics. Such a person need only understand the high-level spec, along with the abstract assertion that the low-level spec implements the high-level spec. This contrasts with the approach of defining desirable/undesirable properties, which of necessity must be understood by anyone wishing to know what the system is supposed to do. The invariants are important only to someone who wants to know **why** the low-level system satisfies the high-level semantics. It is often (but not always) the case that the essence of this why is in the definition of the inductive invariants.

For the particular case of Memoir, we define a single high-level spec but two low-level specs, one for Memoir-Basic and one for Memoir-Opt. We construct two refinements, one that maps from the Memoir-Basic spec to the high-level spec, and one that maps from the Memoir-Opt spec to the Memoir-Basic spec. We prove that the Memoir-Basic spec implements the high-level spec, which requires three inductive invariants. Proving these inductive invariants in turn requires stating and proving two more inductive invariants. We then prove that the Memoir-Opt spec implements the Memoir-Basic spec, which transitively implies that it implements the high-level spec. For this second implementation proof, no inductive invariants are necessary.

### 1.1.3 Assumptions

The proofs herein depend upon several assumptions. Some of these are realized as explicit assumptions using a TLA+ **ASSUME** statement. Others are realized implicitly in the definitions of various actions. The strongest assumptions we make are the following:

**Assumption:** Highly improbable events never occur.

**Realization:**
- The hash function is fully collision-resistant. See the explicit assumptions named *HashCollisionResistant* and *BaseHashValueUnique*.
- The MAC functions are fully collision-resistant and unforgeable. See the explicit assumptions named *MACCollisionResistant* and *MACUnforgeable*.
- Upon restarting, the arbitrary values in the computer’s RAM will not contain an authenticator that is coincidentally equal to an authenticator that could be computed with the symmetric key
Assumption: The untrusted system cannot modify the contents of the NVRAM.

Realization:
- In the Memoir-Basic spec, the only action that changes the value in the NVRAM is \textit{LL1PerformOperation}. See the definitions of \textit{LL1Next} and all actions it disjoins.
- In the Memoir-Opt spec, the only action that changes the value in the NVRAM is \textit{LL2PerformOperation}. See the definitions of \textit{LL2Next} and all actions it disjoins.

Assumption: The SPCR can be modified only by resetting it or extending it. Extending means that the new value is a chained hash of the previous value in the SPCR with another value.

Realization:
- There are only three actions that modify the SPCR: \textit{LL2PerformOperation}, \textit{LL2Restart}, and \textit{LL2CorruptSPCR}. See the definitions of \textit{LL2Next} and all actions it disjoins.
- The \textit{LL2PerformOperation} action extends the SPCR. See the definition of the \textit{LL2PerformOperation} action and the \textit{Successor} operator.
- The \textit{LL2Restart} action resets the SPCR. See the definition of the \textit{LL2Restart} action.
- The \textit{LL2CorruptSPCR} action extends the SPCR. See the definition of the \textit{LL2CorruptSPCR} action.

Assumption: The symmetric key stored in the NVRAM is unknown outside the trusted subsystem.

Realization:
- The only authenticators available to an attacker are (1) those previously returned by Memoir and (2) those the attacker can generate using a symmetric key other than the key stored in the NVRAM. See the definitions of \textit{LL1CorruptRAM} and \textit{LL2CorruptRAM}.

Assumption: The hash barrier stored in the NVRAM is unknown outside the trusted subsystem.

Realization:
- When an attacker extends the SPCR, the value for the extension cannot be constructed as a hash of any value with the hash barrier secret stored in the NVRAM. See the definitions of \textit{LL2CorruptSPCR}.

In addition to the above strong assumptions, we use TLA+ \textbf{ASSUME} statements for several other purposes:

**Type safety of primitives, parameters, and formalisms**
- The primitive operators for hashing, message authentication codes, and symmetric cryptography are assumed to be type-safe. This is asserted by the explicit assumptions \textit{BaseHashValueTypeSafe}, \textit{GenerateMACTypeSafe}, \textit{ValidateMACTypeSafe}, \textit{HashTypeSafe}, \textit{SymmetricEncryptionTypeSafe}, and \textit{SymmetricDecryptionTypeSafe}.
- The service that the Memoir platform executes is assumed to be type-safe, as asserted by the explicit assumptions \textit{ServiceTypeSafe} and \textit{ConstantsTypeSafe}.
- Formalisms needed by the proof are assumed to be type-safe. See the explicit assumptions \textit{HashCardinalityTypeSafe} and \textit{CrazyHashValueTypeSafe}.

**Correctness of primitives**
- The MAC functions are assumed to be complete, meaning that every MAC generated with a key validates correctly with the same key, as asserted by the explicit assumption \textit{MACComplete}.
- The MAC functions are assumed to be consistent, meaning that if a MAC validates correctly with a key, it must have been generated as a MAC with that same key, as asserted by the explicit assumption \textit{MACConsistent}.
- The cryptographic functions are assumed to be correct, meaning that decryption is the inverse of encryption with the same key, as asserted by the explicit assumption \textit{SymmetricCryptoCorrect}.

**Formalisms**
- One consequence of the strong collision-resistance of the hash function is that the result of any hash chain has a well-defined count of hashes that went into its production. We formalize this as the \textit{cardinality} of the hash using the operator \textit{HashCardinality} along with a set of explicit as-
sumptions: HashCardinalityAccumulative, BaseHashCardinalityZero, and InputCardinalityZero.

- When a flag in the Memoir-Opt NVRAM indicates that the SPCR should contain the value BaseHashValue but it in fact contains some other value, we represent the logical value as a formalized CrazyHashValue. This value is assumed to be unequal to any other hash value by the explicit assumption CrazyHashValueUnique.

- The HistorySummariesMatch predicate is defined recursively, but the current version of the prover can neither handle recursive operators nor tractably support proofs using recursive function definitions. Therefore, we define the operator indirectly, by using the explicit assumption HistorySummariesMatchDefinition.

One final—and very important—assumption of this tech report is the correctness of our inductive reasoning. As of this writing, the current version of the TLA+ Proof System is unable to verify the proof step that ties together a base case and an induction step into an inductive proof. We thus depend upon human reasoning skills to ensure that this final step is correct for all proofs that use induction. This includes:

- the use of the Inv1 rule in the proofs of type safety for our three specs and in the proofs of the inductive invariance of four invariants
- the use of the StepSimulation rule in the two implementation proofs
- the final step in the HistorySummariesMatchUniqueLemma, which uses non-temporal inductive reasoning

1.2 Memoir-Basic

Since Memoir is a platform that supports arbitrary services, our high-level spec declares the service to be an undefined function that maps a state and a request to a state and a response. More precisely, the spec partitions the service state into a public portion and a private portion, with the intent that only the private portion need be hidden from the untrusted system by encryption. Thus, the service function takes three arguments—the current public state, the current private state, and an input—and it yields a record with three fields—the new public state, the new private state, and an output. The service also specifies an initial value for the public state, an initial value for the private state, and an initial value for the set of inputs that are available to be processed by the service. This is described precisely in Section 2.1.

1.2.1 High-Level Specification for Memoir-Basic Semantics

The high-level spec for Memoir-Basic semantics contains four state variables: the current public state, the current private state, the set of inputs that are available to be processed by the service, and the set of outputs that the service has been observed to produce. The latter two warrant some explanation.

The set of available inputs is intended to model the fact that, at any given time, some inputs might not be known to the user that invokes the service. For example, if the service is used to redeem cryptographically signed tokens, the user may not know the complete set of valid tokens. The set of available inputs includes the inputs that, at a given moment, are known by the user and thus available to be processed by the service.

The high-level spec includes a variable for the set of outputs observed from the service, because it is important to show that the low-level specs produce a corresponding set of outputs by their actions. It is not enough to show that the public and private states in the refined low-level specs equal the public and private states of the high-level spec, because this would be insufficient to preclude a low-level spec that returns a different set of outputs than are intended by the semantics.

The high-level spec for Memoir-Basic semantics includes two actions. The main action, HLAdvanceService, invokes the service function with arguments of the current public state, the current private state, and an input from the set of available inputs. The output of the service function updates the current public state, the current private state, and the set of outputs observed from the service.

The second action, HLMakeInputAvailable, adds an input to the set of available inputs. This action might, for example, model an out-of-band transaction in which the user pays money in exchange for a signed token, thereby enabling the user to submit a request containing that token.

As argued above (§ 1.1.2), this high-level spec is small and simple enough that it should be easy to determine that its semantics are the right ones. In particular, it is readily apparent that the HLAdvanceService action provides the state continuity desired for the service module.
1.2.2 Memoir-Basic Low-Level Specification

The Memoir-Basic low-level spec contains six state variables. Three of these variables represent concrete state maintained by a Memoir-Basic implementation: contents of the disk, contents of the RAM, and contents of the NVRAM. The only parts of each storage device we model are those of direct relevance to Memoir. For the NVRAM, this is the history summary and symmetric key that Memoir-Basic stores in the NVRAM. For the RAM, this is the values that are exchanged between Memoir and the untrusted system: the current public state and encrypted private state stored by the untrusted system, and the history summary and authenticator that the untrusted system uses to convince Memoir that the current state is valid. For the disk, this is a copy of the contents of the RAM that are stored on the disk for crash-resilience.

The other three variables represent abstractions, two of which are direct analogues of state variables in the high-level spec: the set of inputs that are available to be processed and the set of outputs that have been observed. The third abstract variable is the set of authenticators that the untrusted system has observed to be returned from Memoir. This set is needed as part of the formalism, because an attacker can attempt to re-use any authenticator it has observed Memoir to produce (c.f. § 1.1.3), so the specification needs to track this set to show that its elements are available to an attacker.

The Memoir-Basic low-level spec includes seven actions. The only two actions that model the execution of Memoir-Basic code are \texttt{LL1PerformOperation} and \texttt{LL1RepeatOperation}. The \texttt{LL1PerformOperation} action describes both concrete operations performed by the Memoir-Basic implementation and also abstract operations needed for the formalism. The concrete operations include checking the values in the RAM from the untrusted system against values in NVRAM to ensure correctness and currency, invoking the service function with arguments from the RAM and an input from the set of available inputs, and updating the RAM and NVRAM with new values based on the output of the service function. The abstract operations update the sets of observed outputs and observed authenticators.

The \texttt{LL1RepeatOperation} action behaves similarly to the \texttt{LL1PerformOperation} action, with two main exceptions: First, instead of checking that the state in the RAM is current, it checks that if the state in the RAM were advanced by the given input from the set of available inputs, the resulting state would be current according to the NVRAM. Second, it does not update the NVRAM. Importantly, the \texttt{LL1RepeatOperation} action does update the sets of observed outputs and observed authenticators. This may seem odd, because if the Memoir system is functioning correctly, the \texttt{LL1RepeatOperation} action will not produce an output that it has not previously produced, nor will it produce an authenticator with a meaning other than that of some authenticator it previously produced. However, this is not a property we assume in the definition of the action; it is a property we prove as part of the implementation proof (c.f. the inclusion invariant in § 1.2.4).

A third action, \texttt{LL1MakeInputAvailable}, is an abstract action that is a direct analogue of the high-level spec’s \texttt{HLMakeInputAvailable} action. This action adds an input to the set of available inputs but leaves all concrete state unchanged.

There are three actions that model behavior of the untrusted system. The \texttt{LL1ReadDisk} action reads the state of the disk into the RAM. The \texttt{LL1WriteDisk} action writes the state of the RAM onto the disk. The \texttt{LL1Restart} action models the effect of a system restart by trashing the values in the RAM.

The final Memoir-Basic low-level action is \texttt{LL1CorruptRAM}. This action models an attacker’s ability to put nearly arbitrary values in the RAM before invoking Memoir. As described in Section 1.1.3, because the symmetric key stored in the NVRAM is unknown outside the trusted subsystem, the only authenticators the attacker can put in the RAM are (1) those from the set of authenticators that the untrusted system has observed to be returned from Memoir and (2) those the attacker can generate using a symmetric key other than the key stored in the NVRAM.

1.2.3 Refinement of Memoir-Basic State

The refinement describes how to interpret values of state variables in the Memoir-Basic low-level spec as values of state variables in the high-level spec. There are four high-level variables whose values need to be established through the refinement, two of which are trivial: The high-level variables representing the set of available inputs and the set of observed outputs are asserted by the refinement to respectively equal the corresponding sets from the low-level spec. These are abstract variables, and they have identical meanings across the two specs.

Refining the high-level public and private state is more involved. Intuitively, the only concrete value in the low-level spec that determines the current service state is the history summary in the NVRAM. The values in
the RAM and the disk are irrelevant, because the untrusted system can set these to any values at any time. So, the refinement needs to express that the high-level values of public and private state are values that correspond (in some strong but as yet ill-defined sense) to the history summary in the NVRAM. The way we express this correspondence is by exploiting the set of observed authenticators. Each authenticator expresses a binding between a history summary (such as the one stored in the NVRAM) and a state hash formed from a public and private state. Thus, the refinement asserts that the high-level variables representing the public and private state have any values whose hash is bound to the history summary currently in the NVRAM by some authenticator in the set of observed authenticators. Although it may not be obvious, this assertion uniquely defines the high-level public and private state. We will prove this uniqueness as part of the implementation proof (c.f. the uniqueness invariant in § 1.2.4).

1.2.4 Memoir-Basic Invariants

The proof that the Memoir-Basic low-level spec implements the high-level spec depends upon three inductive invariants: the unforgeability invariant, the inclusion invariant, and the uniqueness invariant, which we collectively refer to as the correctness invariants.

The unforgeability invariant is a somewhat boring invariant. As described above (§ 1.2.2), the definition of the $LL1CorruptRAM$ action constrains the set of authenticators that can be put into the RAM by an attacker. The unforgeability invariant essentially states that the only authenticator values in the RAM are those that satisfy the constraint imposed by the $LL1CorruptRAM$ action. In other words, no other actions violate this constraint. Since one of these other actions, $LL1ReadDisk$, copies the authenticator from the disk into the RAM, we cannot prove the unforgeability invariant directly. Instead, we first prove the extended unforgeability invariant, which applies the authenticator constraint to both the RAM and the disk. The extended unforgeability invariant directly implies the unforgeability invariant.

The inclusion invariant is needed for the implementation proof in the presence of the $LL1RepeatOperation$ action. This invariant essentially states that (1) the output that $LL1RepeatOperation$ will produce is already in the set of observed outputs, and (2) the new authenticator that $LL1RepeatOperation$ will produce authenticates a history state binding that is already being authenticated by some authenticator in the set of observed authenticators. Thus, the $LL1RepeatOperation$ action will not modify these sets in any semantically important way.

The uniqueness invariant states that the the history summary in the NVRAM is bound to only one public and private state by an authenticator in the set of observed authenticators. This invariant is used in the proof that the initial high-level state is correctly defined, in the proofs that the high-level public and private state is not changed by any low-level action that should not change this state, and in the proof that the low-level $LL1PerformOperation$ action implements the behavior of the high-level $HLAdvanceService$ action.

Just as the proof of the unforgeability invariant relies on the extended unforgeability invariant, the proof of the inclusion invariant and the uniqueness invariant also rely on a supplementary invariant, which we call the cardinality invariant. However, unlike the extended unforgeability invariant, the cardinality invariant cannot be proven on its own. Moreover, the inclusion, cardinality, and uniqueness invariants cannot be ordered with respect to each other. The proof of the inclusion invariant inductively depends upon the uniqueness invariant, which in turn depends upon the cardinality invariant, which in turn depends upon the inclusion invariant. We prove these three invariants co-inductively.

The statement of the cardinality invariant relies on a formalism we call the cardinality of a hash, which is the count of hashes that went into the production of any value in the domain of the hash function. The hash cardinality is well-defined because of the strong collision-resistance of the hash function that is assumed (§ 1.1.3) by our proof. The hash cardinality of the base hash value is zero; the hash cardinality of any value not output from the hash function is zero; and the hash cardinality of any output from the hash function is one greater than the hash cardinality of the inputs to the hash function.

The cardinality invariant states that the hash cardinality of the history summary bound by any authenticator in the set of observed authenticators is less than or equal to the hash cardinality of the history summary in the NVRAM. The cardinality invariant inductively supports the proof of the uniqueness invariant, because it allows us to prove that when the $LL1PerformOperation$ action produces a new authenticator, that authenticator binds a history summary that is not bound by any authenticator in the set of observed authenticators, because the new authenticator has a greater hash cardinality than any authenticator in the set. In turn, the uniqueness
invariant inductively supports the proof of the inclusion invariant, because it allows us to prove that when the \textit{LL1PerformOperation} action produces a state hash from the public and private state in the RAM, this equals the state hash defined in the inclusion invariant. Completing the cycle, the inclusion invariant inductively supports the proof of the cardinality invariant, because it allows us to prove that the \textit{LL1RepeatOperation} action makes no semantic change to the set of observed authenticators, and thus there is no change to the set of hash cardinalities represented by this set.

1.2.5 Memoir-Basic Correctness
To prove that the Memoir-Basic low-level spec implements the high-level spec, we prove (1) that the initial state of the low-level spec, under refinement, satisfies the initial state of the high-level spec, and (2) the next-state predicate of the low-level spec, under refinement, satisfies the next-state predicate of the high-level spec. Moreover, the proof of the next-state predicate includes sub-proofs for the following eight implications:

\begin{align*}
\text{unchanged } & LL1\text{Vars} \Rightarrow \text{unchanged } HL\text{Vars} \\
LL1\text{MakeInputAvailable} & \Rightarrow HL\text{MakeInputAvailable} \\
LL1\text{PerformOperation} & \Rightarrow HL\text{AdvanceService} \\
LL1\text{RepeatOperation} & \Rightarrow \text{unchanged } HL\text{Vars} \\
LL1\text{Restart} & \Rightarrow \text{unchanged } HL\text{Vars} \\
LL1\text{ReadDisk} & \Rightarrow \text{unchanged } HL\text{Vars} \\
LL1\text{WriteDisk} & \Rightarrow \text{unchanged } HL\text{Vars} \\
LL1\text{CorruptRAM} & \Rightarrow \text{unchanged } HL\text{Vars}
\end{align*}

In other words:

- A Memoir-Basic stuttering step maps to a high-level stuttering step.
- A Memoir-Basic \textit{LL1MakeInputAvailable} action maps to a high-level \textit{HLMakeInputAvailable} action.
- A Memoir-Basic \textit{LL1PerformOperation} action maps to a high-level \textit{HLAdvanceService} action.
- All other Memoir-Basic actions map to high-level stuttering steps.

The proofs of high-level stuttering all exploit a lemma called the \textit{non-advancement lemma}. This lemma states that, if there is no change to the NVRAM or to the authentication status of any history state binding, then there is no change to the high-level public and private state defined by the refinement. Employing this lemma is completely straightforward for a low-level stuttering step and for the \textit{LL1Restart}, \textit{LL1ReadDisk}, \textit{LL1WriteDisk}, and \textit{LL1CorruptRAM} actions. For the \textit{LL1RepeatOperation} action, this lemma is usable because the inclusion invariant guarantees that \textit{LL1RepeatOperation} does not change the authentication status of any history state binding.

The proof for the \textit{LL1MakeInputAvailable} action is straightforward. The set of available inputs corresponds directly across the two specs, and the non-advancement lemma shows that the high-level state does not change.

The proof for \textit{LL1PerformOperation} uses the uniqueness invariant twice: First, it is used to show that the public and private state in the arguments to the service correspond to the refined high-level state. Second, it is used to show that the service results in a public and private state that corresponds to the refined high-level primed state. Thus, the service processes the same inputs and produces the same outputs as the service in the \textit{HLAdvanceService} action in the high-level spec.

The following table shows which invariants are needed for which action’s proof:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>unforgeability invariant</th>
<th>inclusion invariant</th>
<th>uniqueness invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{LL1Init}</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>UNCHANGED \textit{LL1Vars}</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>\textit{LL1MakeInputAvailable}</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>\textit{LL1PerformOperation}</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>\textit{LL1RepeatOperation}</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>\textit{LL1Restart}</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>\textit{LL1ReadDisk}</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>\textit{LL1WriteDisk}</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>\textit{LL1CorruptRAM}</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
</tbody>
</table>
1.3 Memoir-Opt

The Memoir-Opt system has a different high-level specification than the Memoir-Basic system. In particular, there are actions in the Memoir-Opt system that enable a malicious user of the system to permanently kill the system. Although we cannot prevent the user from killing the system, we wish to ensure that the only undesirable behavior that can happen is the death of the system, meaning that it stops processing inputs and produces no new outputs. Therefore, we modify the high-level spec to add this behavior to the system semantics.

Our approach to proving the correctness of Memoir-Opt is to prove that, under refinement, the Memoir-Opt spec satisfies the Memoir-Basic spec, which transitively implies that it satisfies the high-level spec. However, because Memoir-Opt includes actions that can kill the system, whereas Memoir-Basic does not, it is not possible for the Memoir-Opt spec to satisfy the Memoir-Basic spec we have described so far. Therefore, we modify the Memoir-Basic spec to include an additional action that does not represent any realistic action in a direct implementation of the Memoir-Basic spec. We will prove that this new action in the Memoir-Basic spec maps to a system death in the high-level spec. Then, we will prove that certain actions in the Memoir-Opt spec, under certain conditions, will map to this new action in the Memoir-Basic spec.

1.3.1 Modifications to High-Level Specification for Memoir-Opt Semantics

To support refinement from the Memoir-Opt spec, we modify the high-level spec to add an additional action, which in turn requires adding an additional state variable.

The new state variable is simply a boolean that indicates whether the system is alive. We modify the initial-state predicate to indicate that this variable is true in the initial system state. We also add an enablement condition to the existing high-level \(HL\text{AdvanceService}\) action to require this variable to be true; in other words, the system must be alive for the \(H\text{LAdvanceService}\) action to occur.

The new action we add is \(H\text{LDie}\), which does nothing other than set the new state variable to false. Once the variable becomes false, there is no action that will set it back to true.

1.3.2 Modifications to Memoir-Basic Specification for Memoir-Opt Semantics

To support refinement from the Memoir-Opt spec, we modify the Memoir-Basic spec to add an additional action. This new action, \(LL1\text{RestrictedCorruption}\), does not model any realistic action in a direct implementation of the Memoir-Basic spec. In particular, this action corrupts the history summary stored in the NVRAM, but the TPM prevents any code other than Memoir from writing to the NVRAM.

The purpose of this action is to model the effect on the Memoir-Basic spec that refines from the Memoir-Opt spec when the SPCR in Memoir-Opt is corrupted or inappropriately reset. We need the \(LL1\text{RestrictedCorruption}\) action to be strong enough to enable refinement from actions the Memoir-Opt spec but weak enough to enable refinement to the \(H\text{LDie}\) action in the high-level spec.

We therefore impose two constraints on the corrupted history summary value in the NVRAM. First, to ensure that the \(\text{CardinalityInvariant}\) and \(\text{UniquenessInvariant}\) continue to hold, no authenticator in the set of observed authenticators may validate a history state binding that binds the NVRAM’s history summary to any state hash. Second, to ensure that the \(\text{InclusionInvariant}\) continues to hold, no authenticator in the set of observed authenticators may validate a history state binding that binds any predecessor of the the NVRAM’s history summary to any state hash.

The \(LL1\text{RestrictedCorruption}\) action may also corrupt the state of the RAM in the exact same way the \(LL1\text{Restart}\) action corrupts the RAM, or it may leave the RAM unchanged. Both alternatives are necessary because two different actions in the Memoir-Opt spec refine to the \(LL1\text{RestrictedCorruption}\) action, and although they have the same effect on the NVRAM, they have different effects on the RAM.

1.3.3 Memoir-Opt Low-Level Specification

The Memoir-Opt low-level spec contains seven state variables, three of which represent the same abstractions represented in the Memoir-Basic low-level spec: the set of available inputs, the set of observed outputs, and the set of observed authenticators. The other four state variables represent concrete state maintained by a Memoir-Opt implementation: disk, RAM, NVRAM, and SPCR. The disk and RAM variables are direct analogues of the disk and RAM variables in Memoir-Basic.

The NVRAM variable contains a history summary and symmetric key, just like the Memoir-Basic NVRAM. However, it also stores two additional fields: a hash barrier secret and flag indicating whether an extension is in progress. The SPCR variable, unsurprisingly, models the the SPCR.
The Memoir-Opt low-level spec includes nine actions. Four of these actions are semantically identical to corresponding actions in the Memoir-Basic spec: \textit{LL2MakeInputAvailable}, \textit{LL2ReadDisk}, \textit{LL2WriteDisk}, and \textit{LL2CorruptRAM}. Three other actions, although not identical, are semantically analogous to actions in the Memoir-Basic spec: \textit{LL2PerformOperation}, \textit{LL2RepeatOperation}, and \textit{LL2Restart}. The first two of these action differ from their Memoir-Basic counterparts as described in the Memoir paper.\footnote{The first action, \textit{LL2Restart}, is different only in that it additionally resets the state of the SPCR.}

The remaining two actions have no counterparts in the Memoir-Basic spec. The \textit{LL2TakeCheckpoint} action takes a checkpoint, updating the state of the NVRAM to include the history summary information from the SPCR. The \textit{LL2CorruptSPCR} action models an attacker’s ability to modify the contents of the SPCR by extending it with a nearly arbitrary value. As described in Section 1.1.3, the precise specification of \textit{LL2CorruptSPCR} ensures that (1) the SPCR can only be modified by extending its hash chain, and (2) because the hash barrier stored in the NVRAM is unknown outside the trusted subsystem, the value that extends the PCR cannot incorporate the hash barrier.

### 1.3.4 Refinement of Memoir-Opt State

The refinement describes how to interpret values of state variables in the Memoir-Opt low-level spec as values of state variables in the Memoir-Basic low-level spec. There are three cases for how this interpretation is handled.

The first and simplest case is variables that are directly equal between the two specs. This includes the set of available inputs, the set of observed outputs, and some of the fields of the disk, the RAM, and the NVRAM. For the disk and RAM, the particular fields that are equal across the two specs are the public state and the encrypted private state. For the NVRAM, the symmetric key is equal across the two specs.

The second case is variables that directly “match” across the two specs. This includes the set of observed authenticators and the fields of the disk and RAM that are not (as described above) directly equal, namely the authenticator and history summary. Two authenticators match if they are MACs of history state bindings that bind matching history summaries to equal state hashes. Two history summaries match if they both equal the respective initial history summaries for the two specs or (recursively) if they are both successors (with the same input) of matching history summaries.

The third and most involved case is the history summary in the Memoir-Basic NVRAM, which is refined to match the logical value of the history summary defined by the Memoir-Opt NVRAM and SPCR. The logical value of the anchor is the anchor value in the NVRAM, but the logical extension is the value in the SPCR only if the NVRAM indicates that an extension is in progress; otherwise, the logical extension equals the base hash value. The reason for this is that the \textit{LL2TakeCheckpoint} action clears the flag that indicates whether an extension is in progress, but it does not reset the SPCR to the base hash value. Therefore, between an \textit{LL2TakeCheckpoint} action and an \textit{LL2Restart} action, the logical extension is really the base hash value, even though the SPCR has not yet been reset.

### 1.3.5 Memoir-Opt Correctness

Since we modified the Memoir-Basic low-level spec by adding a new action, we need to state and prove the mapping of this action to the high-level spec. Specifically, in the Memoir-Basic implementation proof, we add an additional sub-proof to the proof of the next-state predicate for the following implication:

\[ LL1\text{RestrictedCorruption} \Rightarrow \text{HLDie} \]

Then, to prove that the Memoir-Opt low-level spec implements the Memoir-Basic low-level spec, we prove (1) that the initial state of the Memoir-Opt spec, under refinement, satisfies the initial state of the Memoir-Basic spec, and (2) the next-state predicate of the Memoir-Opt spec, under refinement, satisfies the next-state predicate of the Memoir-Basic spec. Moreover, the proof of the next-state predicate includes sub-proofs for the following ten implications:

- \textit{UNCHANGED LL2Vars} \Rightarrow \textit{UNCHANGED LL1Vars}
- \textit{LL2MakeInputAvailable} \Rightarrow \textit{LL1MakeInputAvailable}
- \textit{LL2PerformOperation} \Rightarrow \textit{LL1PerformOperation}
- \textit{LL2RepeatOperation} \Rightarrow \textit{LL1RepeatOperation}
- \textit{LL2TakeCheckpoint} \Rightarrow \textit{UNCHANGED LL1Vars}
- \textit{LL2Restart} \Rightarrow
  \begin{align*}
  &\text{IF } \text{LL2NVRAM.extensionInProgress}
  \end{align*}
THEN
    \textit{LL1RestrictedCorruption}
ELSE
    \textit{LL1Restart}
\textit{LL2ReadDisk} \Rightarrow \textit{LL1ReadDisk}
\textit{LL2WriteDisk} \Rightarrow \textit{LL1WriteDisk}
\textit{LL2CorruptRAM} \Rightarrow \textit{LL1CorruptRAM}
\textit{LL2CorruptSPCR} \Rightarrow
    \textbf{IF} \ \textit{LL2NVRAM.extensionInProgress}
    \textbf{THEN}
    \textit{LL1RestrictedCorruption}
    \textbf{ELSE}
    \textit{UNCHANGED LL1Vars}

In other words:

- A Memoir-Opt stuttering step maps to a Memoir-Basic stuttering step.
- Six Memoir-Opt actions directly map to analogous Memoir-Basic actions; these are \textit{LL2MakeInputAvilable}, \textit{LL2PerformOperation}, \textit{LL2RepeatOperation}, \textit{LL2ReadDisk}, \textit{LL2WriteDisk}, and \textit{LL2CorruptRAM}.
- A Memoir-Opt \textit{LL2TakeCheckpoint} action maps to a Memoir-Basic stuttering step.
- A Memoir-Opt \textit{LL2Restart} action maps either to an \textit{LL1RestrictedCorruption} action or to an \textit{LL1Restart} action depending on whether an extension is in progress.
- A Memoir-Opt \textit{LL2CorruptSPCR} action maps either to an \textit{LL1RestrictedCorruption} action or to a Memoir-Basic stuttering step depending on whether an extension is in progress.

In the above list, the final two bullet points merit explanation. It might seem that an \textit{LL2Restart} action should map directly to an \textit{LL1Restart} action, and this is the case under normal operation. In particular, since Memoir-Opt should always perform an \textit{LL2TakeCheckpoint} action immediately prior to restarting, and since the \textit{LL2TakeCheckpoint} action clears the extension-in-progress flag, a subsequent \textit{LL2Restart} action (with no intervening \textit{LL2PerformOperation} action) will map to \textit{LL1Restart}. However, a malicious user can force the system to restart without first taking a checkpoint. If this happens when an extension is in progress, the Memoir system will die. We prove this by the transitive implication:

\textbf{IF} \ \textit{LL2NVRAM.extensionInProgress} \ \textbf{AND} \ \textit{LL2Restart} \ \textbf{THEN} \ \textit{LL1RestrictedCorruption} \ \Rightarrow \ \textbf{HLDie}

Irrespective of whether an extension is in progress, the \textit{LL2Restart} action corrupts the state of the RAM in the exact same way the \textit{LL1Restart} action corrupts the RAM. This is not a problem for the above implication, because we have specified the \textit{LL1RestrictedCorruption} action to allow corruption of the RAM in this exact same manner.

Complementary reasoning applies to the mapping of \textit{LL2CorruptSPCR}. It might seem that this action should map directly to an \textit{LL1RestrictedCorruption} action, since the SPCR holds important data about the service state. However, when the flag in the NVRAM indicates that an extension is not in progress, the state of the SPCR is supposed to equal the base hash value. Therefore, even if the SPCR is corrupted by an \textit{LL2CorruptSPCR} action, the SPCR can be restored to its proper value by an \textit{LL2Restart} action, after which normal operation can resume. Thus, when an extension is not in progress, the \textit{LL2Restart} action does not cause the system to die, which we prove with the following transitive implication:

\textbf{IF} \ \neg \textit{LL2NVRAM.extensionInProgress} \ \textbf{AND} \ \textit{LL2CorruptSPCR} \ \Rightarrow \ \textbf{UNCHANGED LL1Vars} \ \Rightarrow \ \textbf{UNCHANGED HLVars}

Note that if we were specifying liveness as well as safety, this implication would not hold, because the states before and after an \textit{LL2CorruptSPCR} action differ in their liveness, insofar as an \textit{LL2PerformOperation} action can occur beforehand but not afterward, unless it is preceded by an \textit{LL2Restart} action.

1.4 Organization

The remainder of this tech report includes the following items:

High-level spec: There is a single high-level spec that defines the semantics of the Memoir system. It includes both the basic semantics of the Memoir-Basic implementation and also the additional semantics of the Memoir-Opt implementation.
Low-level primitives: The low-level specifications make use of several primitives, namely a hash function, MAC functions, and symmetric cryptography. These functions are specified by undefined operators, along with explicit assumptions about the guarantees made by the operators.

Memoir-Basic low-level spec: The Memoir-Basic spec describes how a Memoir-Basic implementation behaves, in terms of input, output, and operations on the disk, RAM, and NVRAM. This spec also includes an additional action (which is not part of a a Memoir-Basic implementation) that supports refinement from the Memoir-Opt specification.

Memoir-Opt low-level spec: The Memoir-Opt spec describes how a Memoir-Opt implementation behaves, in terms of input, output, and operations on the disk, RAM, NVRAM, and SPCR.

Refinements: There are two refinements. One describes the mapping of Memoir-Basic state to high-level state. The other describes the mapping of Memoir-Opt state to Memoir-Basic state.

Invariants: There are five invariants maintained by the Memoir-Basic specification, three of which are needed by the proof that the Memoir-Basic spec satisfies the high-level spec. There are no invariants needed for the proof that the Memoir-Opt spec satisfies the Memoir-Basic spec.

Type-safety theorems: TLA+ is an untyped language, so we state and prove the types of all variables maintained by each of the three specs.

Invariance theorems: The invariants are proven using a temporal inductive proof rule called Inv1. To employ this rule, we prove that each invariant is satisfied in the initial system state, and we prove that each valid action preserves the invariant.

Implementation theorems: There are two implementation theorems: The first states that Memoir-Basic implements the high-level spec, and the second states that Memoir-Opt implements Memoir-Basic. Each implementation is proven using a temporal inductive proof rule called StepSimulation. To employ this rule, we prove that the initial state of each lower-level spec, under refinement, satisfies the initial state of the corresponding higher-level spec, and that each lower-level action corresponds to a higher-level action.

Ancillary Lemmas: There are quite a few lemmas that support the invariance proofs and/or the implementation proofs.

These items are partitioned into TLA+ organizational structures called modules, which group related items together. There are 21 modules in this set of specifications and proofs. Within Sections 2–4, each subsection corresponds to one module.

1.4.1 Organization of TLA+ Modules
Multiple modules are combined by the process of extension*. Each module can extend the declarations and definitions of one or more other modules. The Memoir modules are organized into a linear chain, wherein each of the following modules extends the one before it:

- MemoirCommon—declarations common to high- and low-level specs
- MemoirHLSpecification—specification of the high-level system (semantics)
- MemoirHLTypeSafety—proof of type safety of the high-level spec
- MemoirLLPrimitives—primitives used by the low-level systems
- MemoirLL1Specification—specification of the Memoir-Basic system
- MemoirLL1Refinement—refinement 1: mapping Memoir-Basic state to high-level state
- MemoirLL1TypeLemmas—proofs of lemmas relating to types in the Memoir-Basic Spec
- MemoirLL1TypeSafety—proof of type safety of the Memoir-Basic spec
- MemoirLL1CorrectnessInvariants—invariants needed to prove Memoir-Basic implementation
- MemoirLL1SupplementalInvariants—invariants needed to prove Memoir-Basic invariance
- MemoirLL1InvarianceLemmas—proofs of lemmas that support Memoir-Basic invariance proofs

*This use of the term ‘extension’ is completely unrelated to the ‘extension’ performed on the SPCR. The name collision is unfortunate but unavoidable, since both uses predate our work.
• MemoirLL1UnforgeabilityInvariance—proof of unforgeability invariance in Memoir-Basic
• MemoirLL1InclCardUniqInvariance—proof of inclusion, cardinality, and uniqueness co-invariance in Memoir-Basic
• MemoirLL1Implementation—proof that Memoir-Basic spec implements high-level spec
• MemoirLL2Specification—specification of the Memoir-Opt system
• MemoirLL2Refinement—refinement 2: mapping Memoir-Opt state to Memoir-Basic state
• MemoirLL2TypeLemmas—proofs of lemmas relating to types in the Memoir-Opt spec
• MemoirLL2TypeSafety—proof of type safety of the Memoir-Opt spec
• MemoirLL2RefinementLemmas—proofs of lemmas relating to the Memoir-Opt refinement
• MemoirLL2ImplementationLemmas—proofs of lemmas relating to the Memoir-Opt implementation
• MemoirLL2Implementation—proof that Memoir-Opt spec implements Memoir-Basic spec

This organization largely reflects the order in which the modules were developed. More importantly, the organization places each proof roughly as early in the chain as possible, so that only the items it depends on come before it.

However, within this document, we present the modules in a different order that is more conducive to reading. First, we present the modules pertaining to the specification of the system’s behavior and its implementation (Section 2, beginning on page 18). Second, we present the modules pertaining to the refinement of one spec to another, as well as modules describing invariants maintained by the specs (Section 3, beginning on page 40). Third, we present the modules that contain proofs (Section 4, beginning on page 51).

1.4.2 Index of Declarations
Following is a complete list of all declarations in this set of formal specs and proofs, along with the page on which the declaration can be found. The declarations are partitioned into those of constants, variables, definitions, assumptions, and theorems.

Constants

<table>
<thead>
<tr>
<th>Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseHashValue</td>
<td>22</td>
</tr>
<tr>
<td>CrazyHashValue</td>
<td>45</td>
</tr>
<tr>
<td>DeadPrivateState</td>
<td>19</td>
</tr>
<tr>
<td>DeadPublicState</td>
<td>19</td>
</tr>
<tr>
<td>GenerateMAC(<em>,</em>)</td>
<td>22</td>
</tr>
<tr>
<td>Hash(<em>,</em>)</td>
<td>22</td>
</tr>
<tr>
<td>HashCardinality(_)</td>
<td>49</td>
</tr>
<tr>
<td>HashType</td>
<td>22</td>
</tr>
<tr>
<td>HistorySummariesMatch(<em>,</em>,_)</td>
<td>43</td>
</tr>
<tr>
<td>InitialAvailableInputs</td>
<td>19</td>
</tr>
<tr>
<td>InitialPrivateState</td>
<td>19</td>
</tr>
<tr>
<td>InitialPublicState</td>
<td>19</td>
</tr>
<tr>
<td>InputType</td>
<td>19</td>
</tr>
<tr>
<td>MACType</td>
<td>22</td>
</tr>
<tr>
<td>OutputType</td>
<td>19</td>
</tr>
<tr>
<td>PrivateStateEncType</td>
<td>22</td>
</tr>
<tr>
<td>PrivateStateType</td>
<td>19</td>
</tr>
<tr>
<td>PublicStateType</td>
<td>19</td>
</tr>
<tr>
<td>Service(<em>,</em>,_)</td>
<td>19</td>
</tr>
<tr>
<td>SymmetricDecrypt(<em>,</em>)</td>
<td>22</td>
</tr>
<tr>
<td>SymmetricEncrypt(<em>,</em>)</td>
<td>22</td>
</tr>
<tr>
<td>SymmetricKeyType</td>
<td>22</td>
</tr>
<tr>
<td>ValidateMAC(<em>,</em>,_)</td>
<td>22</td>
</tr>
</tbody>
</table>
Variables

HLAlive ................................................................. 20
HLAvailableInputs .................................................. 20
HLObservedOutputs ................................................ 20
HLPrivateState ..................................................... 20
HLPublicState .......................................................... 20
LL1AvailableInputs .................................................. 24
LL1Disk ............................................................... 24
LL1NVRAM ............................................................. 24
LL1ObservedAuthenticators ........................................ 24
LL1ObservedOutputs ................................................ 24
LL1RAM ............................................................... 24
LL2AvailableInputs .................................................. 31
LL2Disk ............................................................... 31
LL2NVRAM ............................................................. 32
LL2ObservedAuthenticators ........................................ 31
LL2ObservedOutputs ................................................ 31
LL2RAM ............................................................... 32
LL2SPCR ............................................................... 32

Definitions

AuthenticatorSetsMatch(·, ·, ·) ........................................ 44
AuthenticatorsMatch(·, ·, ·) .......................................... 44
CardinalityInvariant ................................................... 49
Checkpoint(·) ............................................................ 32
CorrectnessInvariants .................................................. 48
ExtendedUnforgeabilityInvariant ..................................... 49
HashDomain ............................................................ 22
HistorySummariesMatchRecursion(·, ·, ·) ......................... 43
HistorySummaryType .................................................. 31
HLAdvanceService ..................................................... 20
HLDie ................................................................. 21
HLInit ................................................................. 21
HLMakeInputAvailable ................................................ 20
HLNext ............................................................... 21
HLSpec ............................................................... 21
HLTypeInvariant ..................................................... 20
HLVars ............................................................... 20
InclusionInvariant .................................................... 47
LL1CorruptRAM ......................................................... 28
LL1HistoryState.BindingAuthenticated(·) ......................... 41
LL1Init ............................................................... 29
LL1MakeInputAvailable ................................................ 25
LL1Next ............................................................... 30
LL1NVRAMHistorySummaryUncorrupted ......................... 41
LL1PerformOperation ................................................ 25
LL1ReadDisk ............................................................. 27
LL1Refinement .......................................................... 41
LL1RepeatOperation .................................................. 26
LL1Restart ............................................................. 27
LL1RestrictedCorruption ............................................... 28
LL1Spec ............................................................... 30
LL1SubtypeImplication ............................................... 54
<table>
<thead>
<tr>
<th>Assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseHashCardinalityZero</td>
</tr>
<tr>
<td>BaseHashValueUnique</td>
</tr>
<tr>
<td>CryptoHashValueUnique</td>
</tr>
<tr>
<td>CryptoHashValueTypeSafe</td>
</tr>
<tr>
<td>UniquenessInvariant</td>
</tr>
<tr>
<td>HashCollisionResistant</td>
</tr>
<tr>
<td>HashCardinalityTypeSafe</td>
</tr>
<tr>
<td>GenerateMACTypeSafe</td>
</tr>
<tr>
<td>MACCollisionResistant</td>
</tr>
<tr>
<td>HistorySummariesMatchDefinition</td>
</tr>
<tr>
<td>MACComplete</td>
</tr>
<tr>
<td>MACConsistent</td>
</tr>
<tr>
<td>MACUnderflow</td>
</tr>
<tr>
<td>UntrustedStorageType Vars</td>
</tr>
<tr>
<td>TrustedStorageType Vars</td>
</tr>
<tr>
<td>TakeCheckpoint</td>
</tr>
<tr>
<td>Restart</td>
</tr>
<tr>
<td>ReadDisk</td>
</tr>
<tr>
<td>CorruptSPCR</td>
</tr>
<tr>
<td>CorruptRAM</td>
</tr>
<tr>
<td>WriteDisk</td>
</tr>
<tr>
<td>Init</td>
</tr>
<tr>
<td>MakeInputAvailable</td>
</tr>
</tbody>
</table>

The image contains a page with text that is not clearly visible due to the resolution or angle of the scan. It appears to be a table or list related to assumptions or properties, possibly in a technical or academic context. The text includes terms like `BaseHashCardinalityZero`, `BaseHashValueUnique`, `CryptoHashValueUnique`, `CryptoHashValueTypeSafe`, `UniquenessInvariant`, `HashCollisionResistant`, `HashCardinalityTypeSafe`, `GenerateMACTypeSafe`, `MACCollisionResistant`, `HistorySummariesMatchDefinition`, `MACComplete`, `MACConsistent`, and `MACUnderflow`. There are also references to `UntrustedStorageType Vars` and `TrustedStorageType Vars`. The page number is 15, and the text seems to be related to security or cryptographic properties.
Theorems

AUTHENTICATOR_GENERATED_LEMMA ............................................................... 252
AUTHENTICATOR_INDEX_SET_LEMMA ................................................................. 250
AUTHENTICATOR_SETS_MATCH_UNIQUE_LEMMA ................................................... 248
AUTHENTICATORS_MATCH_DEFS_TYPE_SAFE_LEMMA ............................................ 210
AUTHENTICATORS_MATCH_UNIQUE_LEMMA ......................................................... 245
AUTHENTICATOR_VALIDATED_LEMMA ................................................................ 257
CARDINALITY_INVARIANT_DEFS_TYPE_SAFE_LEMMA ................................................. 65
CARDINALITY_UNCHANGED_LEMMA ................................................................... 101
CHECKPOINT_DEFS_TYPE_SAFE_LEMMA ............................................................. 192
CHECKPOINT_HAS_BASE_EXTENSION_LEMMA ....................................................... 234
CHECKPOINT_TYPE_SAFE ................................................................................... 193
CORRECTNESS_INVARIANCE ............................................................................. 156
EXTENDED_UNFORGEABILITY_INVARIANCE ...................................................... 107
GEOQORLT ......................................................................................................... 82
HISTORY_SUMMARIES_MATCH ACROSS_CHECKPOINT_LEMMA .............................. 259
HISTORY_SUMMARIES_MATCH_UNIQUE_LEMMA ................................................... 236
HISTORY_SUMMARY_RECORD_COMPOSITION_LEMMA .......................................... 230
HISTORY_TYPE_SAFE ......................................................................................... 52
INCLUSION_CARDINALITY_UNIQUENESS_INVARIANCE ...................................... 120
INCLUSION_INVARIANT_DEFS_TYPE_SAFE_LEMMA ............................................. 63
INCLUSION_INVARIANT_UNCHANGED_LEMMA .................................................... 96
LEQ_TRANSITIVE .............................................................................................. 82
LL1_DISK_RECORD_COMPOSITION_LEMMA ......................................................... 230
LL1_DISK_UNFORGEABILITY_UNCHANGED_LEMMA ............................................ 94
LL1_IMPLEMENTATION ....................................................................................... 162
LL1_INIT_DEFS_TYPE_SAFE_LEMMA ................................................................. 56
LL1_NV_RAM_HISTORY_SUMMARY_UNCORRUPTED_EQUALS_HLAIVE_LEMMA ........ 162
LL1_NV_RAM_HISTORY_SUMMARY_UNCORRUPTED_UNCHANGED_LEMMA ............ 83
LL1_NV_RAM_RECORD_COMPOSITION_LEMMA ..................................................... 233
LL1_PERFORM_OPERATION_DEFS_TYPE_SAFE_LEMMA ........................................ 58
LL1_RAM_RECORD_COMPOSITION_LEMMA .......................................................... 232
LL1_RAM_UNFORGEABILITY_UNCHANGED_LEMMA .............................................. 92
LL1_REFINEMENT_DEFS_TYPE_SAFE_LEMMA ..................................................... 67
LL1_REFINEMENT_PRIME_DEFS_TYPE_SAFE_LEMMA .......................................... 67
LL1_REPEAT_OPERATION_DEFS_TYPE_SAFE_LEMMA ............................................ 60
LL1_REPEAT_OPERATION_UNCHANGED_AUTHENTICATED_HISTORICAL_STATE_BINDINGS_LEMMA ................................. 88
LL1_REPEAT_OPERATION_UNCHANGED_OBSERVED_OUTPUTS_LEMMA ..................... 86
LL1_SUBTYPE_IMPICATION_LEMMA ................................................................... 54
LL1_TYPE_SAFE .................................................................................................. 69
LL2_CORRUPT_SPCR_DEFS_TYPE_SAFE_LEMMA .................................................. 209
LL2_IMPLEMENTATION ....................................................................................... 284
LL2_INIT_DEFS_TYPE_SAFE_LEMMA ................................................................. 198
LL2_NV_RAM_LOGICAL_HISTORICAL_SUMMARY_TYPE_SAFE ............................ 249
LL2_PERFORM_OPERATION_DEFS_TYPE_SAFE_LEMMA .......................................... 200
LL2_REPEAT_OPERATION_DEFS_TYPE_SAFE_LEMMA ............................................ 204
LL2_SUBTYPE_IMPICATION_LEMMA ................................................................... 195
LL2_TAKE_CHECKPOINT_DEFS_TYPE_SAFE_LEMMA ............................................. 209
LL2_TYPE_SAFE .................................................................................................. 213
NON_ADVANCEMENT_LEMMA ............................................................................ 157
SUCCESSOR_DEFS_TYPE_SAFE_LEMMA .............................................................. 193
SUCCESSOR_HAS_NON_BASE_EXTENSION_LEMMA .............................................. 235
<table>
<thead>
<tr>
<th>Lemma Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SuccessorTypeSafe</td>
<td>195</td>
</tr>
<tr>
<td>SymmetricKeyConstantLemma</td>
<td>82</td>
</tr>
<tr>
<td>TypeSafetyRefinementLemma</td>
<td>211</td>
</tr>
<tr>
<td>UnchangedAuthenticatedHistoryStateBindingsLemma</td>
<td>105</td>
</tr>
<tr>
<td>UnchangedAvailableInputsLemma</td>
<td>271</td>
</tr>
<tr>
<td>UnchangedDiskAuthenticatorLemma</td>
<td>275</td>
</tr>
<tr>
<td>UnchangedDiskHistorySummaryLemma</td>
<td>274</td>
</tr>
<tr>
<td>UnchangedDiskLemma</td>
<td>276</td>
</tr>
<tr>
<td>UnchangedDiskPrivateStateEncLemma</td>
<td>274</td>
</tr>
<tr>
<td>UnchangedDiskPublicStateLemma</td>
<td>273</td>
</tr>
<tr>
<td>UnchangedNVRAMHistorySummaryLemma</td>
<td>281</td>
</tr>
<tr>
<td>UnchangedNVRAMLemma</td>
<td>282</td>
</tr>
<tr>
<td>UnchangedNVRAMSymmetricKeyLemma</td>
<td>282</td>
</tr>
<tr>
<td>UnchangedObservedAuthenticatorsLemma</td>
<td>272</td>
</tr>
<tr>
<td>UnchangedObservedOutputsLemma</td>
<td>272</td>
</tr>
<tr>
<td>UnchangedRAMAuthenticatorLemma</td>
<td>279</td>
</tr>
<tr>
<td>UnchangedRAMHistorySummaryLemma</td>
<td>278</td>
</tr>
<tr>
<td>UnchangedRAMLemma</td>
<td>280</td>
</tr>
<tr>
<td>UnchangedRAMPrivateStateEncLemma</td>
<td>277</td>
</tr>
<tr>
<td>UnchangedRAMPublicStateLemma</td>
<td>277</td>
</tr>
<tr>
<td>UnforgeabilityInvariance</td>
<td>119</td>
</tr>
<tr>
<td>UniquenessInvariantDefsTypeSafeLemma</td>
<td>66</td>
</tr>
<tr>
<td>UniquenessUnchangedLemma</td>
<td>102</td>
</tr>
</tbody>
</table>
2. SPECIFICATIONS

This section presents TLA+ modules pertaining to the specification of the system’s behavior and its implementation. This includes common declarations, definitions of low-level primitives, and specifications for the high-level spec and both low-level specs.

As a guide to understanding the impact of the actions in each spec, the following tables show which state variables are read and/or written by each action. To keep these tables from being useless, we employ definitions of “read” and “written” that are slightly non-obvious with respect to the formal specification. In particular, we ignore the fact that the UNCHANGED predicate both “reads” and “writes” a variable, insofar as it specifies that the primed state of the variable equals the unprimed state of that variable.

<table>
<thead>
<tr>
<th>Action</th>
<th>HLAlive</th>
<th>HLAvailable Inputs</th>
<th>HLObserved Outputs</th>
<th>HLPublic State</th>
<th>HLPriate State</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLMakeInputAvailable</td>
<td>-</td>
<td>R/W</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HLAdvanceService</td>
<td>R</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
</tr>
<tr>
<td>HLDie</td>
<td>W</td>
<td>-</td>
<td>-</td>
<td>W</td>
<td>W</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action</th>
<th>LL1Available Inputs</th>
<th>LL1Observed Outputs</th>
<th>LL1Observed Authenticators</th>
<th>LL1 Disk</th>
<th>LL1 RAM</th>
<th>LL1 NVRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>LL1MakeInputAvailable</td>
<td>R/W</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LL1PerformOperation</td>
<td>R</td>
<td>R/W</td>
<td>R/W</td>
<td>-</td>
<td>R/W</td>
<td>R/W</td>
</tr>
<tr>
<td>LL1RepeatOperation</td>
<td>R</td>
<td>R/W</td>
<td>R/W</td>
<td>-</td>
<td>R/W</td>
<td>R</td>
</tr>
<tr>
<td>LL1Restart</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>W</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>LL1ReadDisk</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>R</td>
<td>W</td>
<td>-</td>
</tr>
<tr>
<td>LL1WriteDisk</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>W</td>
<td>R</td>
<td>-</td>
</tr>
<tr>
<td>LL1CorruptRAM</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>W</td>
<td>R</td>
</tr>
<tr>
<td>LL1RestrictedCorruption</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>R/W</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action</th>
<th>LL2Available Inputs</th>
<th>LL2Observed Outputs</th>
<th>LL2Observed Authenticators</th>
<th>LL2 Disk</th>
<th>LL2 RAM</th>
<th>LL2 NVRAM</th>
<th>LL2 SPCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LL2MakeInputAvailable</td>
<td>R/W</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LL2PerformOperation</td>
<td>R</td>
<td>R/W</td>
<td>R/W</td>
<td>-</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
</tr>
<tr>
<td>LL2RepeatOperation</td>
<td>R</td>
<td>R/W</td>
<td>R/W</td>
<td>-</td>
<td>R/W</td>
<td>R</td>
<td>-</td>
</tr>
<tr>
<td>LL2TakeCheckpoint</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>R/W</td>
<td>R</td>
<td>R</td>
<td>R/W</td>
</tr>
<tr>
<td>LL2Restart</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>W</td>
<td>R</td>
<td>R/W</td>
<td>-</td>
</tr>
<tr>
<td>LL2ReadDisk</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>R</td>
<td>W</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LL2WriteDisk</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>W</td>
<td>R</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LL2CorruptRAM</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>W</td>
<td>R</td>
<td>-</td>
</tr>
<tr>
<td>LL2CorruptSPCR</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>R</td>
<td>R/W</td>
<td>-</td>
</tr>
</tbody>
</table>
2.1 Declarations Common to High- and Low-Level Specs

This module defines some basic constants used by both the high-level and low-level specs.

A developer that wishes to use Memoir is expected to provide a service implementation that (1) operates on some application-specific input, (2) produces some application-specific output, and (3) maintains some application-specific public and private state. The developer also specifies an initial public and private state for the service. The service is assumed to be type-safe.

The one non-obvious aspect of this module is the constant `InitialAvailableInputs`, which will be explained in the comments relating to the high-level spec.

```latex

declare

initial available inputs

constant

InputType

constant

OutputType

constant

PublicStateType

constant

PrivateStateType

constant

newPublicState : PublicStateType,
newPrivateState : PrivateStateType,
output : OutputType

constant

Service(., ., .)

assume

∀ input ∈ InputType, publicState ∈ PublicStateType, privateState ∈ PrivateStateType :
Service(publicState, privateState, input) ∈ ServiceResultType

constant

InitialAvailableInputs
constant

InitialPublicState
constant

InitialPrivateState
constant

DeadPublicState
constant

DeadPrivateState

assume

InitialAvailableInputs ⊆ InputType
InitialPublicState ∈ PublicStateType
InitialPrivateState ∈ PrivateStateType
DeadPublicState ∈ PublicStateType
DeadPrivateState ∈ PrivateStateType
```
This module defines the high-level behavior of Memoir. There are three actions:

\[ \text{HLMakeInputAvailable} \]
\[ \text{HLAdvanceService} \]
\[ \text{HLDie} \]

**EXTRA**

**EXTENDS** MemoirCommon

**VARIABLE** HLAlive

**VARIABLE** HLAvailableInputs

**VARIABLE** HLObservedOutputs

**VARIABLE** HLPublicState

**VARIABLE** HLPublicState

**HLTypeInvariant**
\[ \Delta \equiv \]
\[ \land \text{HLAlive} \in \text{BOOLEAN} \]
\[ \land \text{HLAvailableInputs} \subseteq \text{InputType} \]
\[ \land \text{HLObservedOutputs} \subseteq \text{OutputType} \]
\[ \land \text{HLPublicState} \in \text{PublicStateType} \]
\[ \land \text{HLPublicState} \in \text{PrivateStateType} \]

**HLVars**
\[ \Delta \equiv \langle \text{HLAlive}, \text{HLAvailableInputs}, \text{HLObservedOutputs}, \text{HLPublicState}, \text{HLPublicState} \rangle \]

The **HLAdvanceService** action is not allowed to take just any input from **InputType**. It may only take an input from the set **HLAvailableInputs**. This models the fact that some inputs might not be known to the user that invokes the service. For example, the service might be used to redeem cryptographically signed tokens, and the user does not initially know the complete set of valid tokens. The user might, for example, have to pay money to retrieve a token from a server, and when the user does so, this corresponds to the action **HLMakeInputAvailable**, which puts the input that includes this token into the set of **HLAvailableInputs**.

**HLMakeInputAvailable**
\[ \Delta \equiv \]
\[ \exists \text{input} \in \text{InputType} : \]
\[ \land \text{input} \notin \text{HLAvailableInputs} \]
\[ \land \text{HLAvailableInputs}' = \text{HLAvailableInputs} \cup \{\text{input}\} \]
\[ \land \text{UNCHANGED} \text{HLObservedOutputs} \]
\[ \land \text{UNCHANGED} \text{HLAlive} \]
\[ \land \text{UNCHANGED} \text{HLPublicState} \]
\[ \land \text{UNCHANGED} \text{HLPublicState} \]

The high-level behavior is a service. There is one main action, which is **HLAdvanceService**. This action takes some input, invokes the developer-supplied service with this input and the current public and private state, updates the public and private state accordingly, and adds the output to the set of observed outputs.

**HLAdvanceService**
\[ \Delta \equiv \]
\[ \exists \text{input} \in \text{HLAvailableInputs} : \]
\[ \text{LET} \]
\[ \text{hlSResult} \Delta \equiv \text{Service}(\text{HLPublicState}, \text{HLPublicState}, \text{input}) \]
\[ \text{IN} \]
\[ \land \text{HLAlive} = \text{TRUE} \]
\[ \land \text{HLPublicState}' = \text{hlSResult}.\text{newPublicState} \]
\[ \land \text{HLPublicState}' = \text{hlSResult}.\text{newPrivateState} \]
\[ \land \text{HLObservedOutputs}' = \text{HLObservedOutputs} \cup \{\text{hlSResult}.\text{output}\} \]
The high-level spec includes the \(\text{HLDie}\) action, which kills the system. This is necessary because, in the low-level specs, it is possible for an adversary to perform an action that causes the system to no longer function. Therefore, we must admit a corresponding action in the high-level spec. Importantly, the \(\text{HLDie}\) action does not change the set of observed outputs, so it cannot be used to trick the system into providing an output it would not otherwise be willing to provide.

\[
\begin{align*}
\text{HLDie} & \triangleq \\
& \wedge \text{HL Alive}' = \text{false} \\
& \wedge \text{UNCHANGED } \text{HL Available Inputs} \\
& \wedge \text{UNCHANGED } \text{HLObserved Outputs} \\
& \wedge \text{HL Public State}' = \text{Dead Public State} \\
& \wedge \text{HL Private State}' = \text{Dead Private State}
\end{align*}
\]

\[
\begin{align*}
\text{HLInit} & \triangleq \\
& \wedge \text{HL Alive} = \text{true} \\
& \wedge \text{HL Available Inputs} = \text{Initial Available Inputs} \\
& \wedge \text{HLObserved Outputs} = \{} \\
& \wedge \text{HL Public State} = \text{Initial Public State} \\
& \wedge \text{HL Private State} = \text{Initial Private State}
\end{align*}
\]

\[
\begin{align*}
\text{HLNext} & \triangleq \\
& \vee \text{HL Make Input Available} \\
& \vee \text{HL Advance Service} \\
& \vee \text{HLDie}
\end{align*}
\]

\[
\begin{align*}
\text{HLSpec} & \triangleq \text{HLInit} \land \Box [\text{HLNext}]_{\text{HLVars}}
\end{align*}
\]
2.3 Primitives Used by the Low-Level Systems

This module defines primitives that are used by the low-level specs. The primitives include a hash function, MAC functions, and symmetric crypto functions, along with their associated types. The module also asserts assumptions about the properties of these functions.

EXTENDS MemoirHLTypeSafety

The low-level specs make use of three primitives: a secure hash, a MAC (message authentication code), and symmetric cryptography.

CONSTANT HashType
CONSTANT MACType
CONSTANT SymmetricKeyType
CONSTANT PrivateStateEncType

CONSTANT Hash(_, _) : HashType
CONSTANT GenerateMAC(_, _) : MACType
CONSTANT ValidateMAC(_, _, _) : MACType
CONSTANT SymmetricEncrypt(_, _) : SymmetricKeyType
CONSTANT SymmetricDecrypt(_, _) : SymmetricKeyType

The hash function has a somewhat strange signature. It accepts two arguments, rather than one. The reason for this is so that we can construct hash chains. Alternatively, we could have written the spec with a conventional single-argument hash function and a two-argument concatenation function, but this would have added complexity for no real benefit.

We assume a base hash value, which in a real implementation, might just be the value zero.

CONSTANT BaseHashValue

The domain of the hash function is hashes, inputs, public states, and encrypted private states. These are the only types we need to hash.

HashDomain ≝ UNION { HashType, InputType, PublicStateType, PrivateStateEncType }

The base hash value is a valid hash value, and it cannot be produced by hashing any other value.

ASSUME BaseHashValueTypeSafe ≝ BaseHashValue ∈ HashType

ASSUME BaseHashValueUnique ≝ ∀ hashInput1, hashInput2 ∈ HashDomain : Hash(hashInput1, hashInput2) ≠ BaseHashValue

The hash function is assumed to be type-safe and collision-resistant. In this spec, we define collision resistance in a very strong sense, namely that there are no different inputs that will hash to the same output. Although a real implementation of a hash function cannot satisfy this, cryptographically secure hash functions are expected to practically satisfy such a condition.

ASSUME HashTypeSafe ≝ ∀ hashInput1, hashInput2 ∈ HashDomain : Hash(hashInput1, hashInput2) ∈ HashType
ASSUME \( \text{HashCollisionResistant} \triangleq \)
\[ \forall \text{hashInput}_1a, \text{hashInput}_2a, \text{hashInput}_1b, \text{hashInput}_2b \in \text{HashDomain} : \]
\[ \text{Hash}(\text{hashInput}_1a, \text{hashInput}_2a) = \text{Hash}(\text{hashInput}_1b, \text{hashInput}_2b) \Rightarrow \]
\[ \land \text{hashInput}_1a = \text{hashInput}_1b \]
\[ \land \text{hashInput}_2a = \text{hashInput}_2b \]

The MAC functions are assumed to be type-safe, complete, consistent, unforgeable, and collision-resistant.

ASSUME \( \text{GenerateMACTypeSafe} \triangleq \)
\[ \forall \text{key} \in \text{SymmetricKeyType}, \text{hash} \in \text{HashType} : \]
\[ \text{GenerateMAC(\text{key}, \text{hash})} \in \text{MACType} \]

ASSUME \( \text{ValidateMACTypeSafe} \triangleq \)
\[ \forall \text{key} \in \text{SymmetricKeyType}, \text{hash} \in \text{HashType}, \text{mac} \in \text{MACType} : \]
\[ \text{ValidateMAC(\text{key}, \text{hash}, \text{mac})} \in \text{boolean} \]

ASSUME \( \text{MACComplete} \triangleq \)
\[ \forall \text{key} \in \text{SymmetricKeyType}, \text{hash} \in \text{HashType} : \]
\[ \text{ValidateMAC(\text{key}, \text{hash}, \text{GenerateMAC(\text{key}, \text{hash})})} = \text{TRUE} \]

ASSUME \( \text{MACConsistent} \triangleq \)
\[ \forall \text{key} \in \text{SymmetricKeyType}, \text{hash} \in \text{HashType}, \text{mac} \in \text{MACType} : \]
\[ \text{ValidateMAC(\text{key}, \text{hash}, \text{mac})} \Rightarrow \text{mac} = \text{GenerateMAC(\text{key}, \text{hash})} \]

ASSUME \( \text{MACUnforgeable} \triangleq \)
\[ \forall \text{key}_1, \text{key}_2 \in \text{SymmetricKeyType}, \text{hash}_1, \text{hash}_2 \in \text{HashType} : \]
\[ \text{ValidateMAC(\text{key}_1, \text{hash}_1, \text{GenerateMAC(\text{key}_2, \text{hash}_2)})} \Rightarrow \text{key}_1 = \text{key}_2 \]

ASSUME \( \text{MACCollisionResistant} \triangleq \)
\[ \forall \text{key}_1, \text{key}_2 \in \text{SymmetricKeyType}, \text{hash}_1, \text{hash}_2 \in \text{HashType} : \]
\[ \text{ValidateMAC(\text{key}_1, \text{hash}_1, \text{GenerateMAC(\text{key}_2, \text{hash}_2)})} \Rightarrow \text{hash}_1 = \text{hash}_2 \]

The symmetric-crypto functions are assumed to be type-safe. They are also assumed to be correct, meaning that decryption is the inverse of encryption, given the same crypto key.

ASSUME \( \text{SymmetricEncryptionTypeSafe} \triangleq \)
\[ \forall \text{key} \in \text{SymmetricKeyType}, \text{privateState} \in \text{PrivateStateType} : \]
\[ \text{SymmetricEncrypt(\text{key}, \text{privateState})} \in \text{PrivateStateEncType} \]

ASSUME \( \text{SymmetricDecryptionTypeSafe} \triangleq \)
\[ \forall \text{key} \in \text{SymmetricKeyType}, \text{privateStateEnc} \in \text{PrivateStateEncType} : \]
\[ \text{SymmetricDecrypt(\text{key}, \text{privateStateEnc})} \in \text{PrivateStateType} \]

ASSUME \( \text{SymmetricCryptoCorrect} \triangleq \)
\[ \forall \text{key} \in \text{SymmetricKeyType}, \text{privateState} \in \text{PrivateStateType} : \]
\[ \text{SymmetricDecrypt(\text{key}, \text{SymmetricEncrypt(\text{key}, \text{privateState})})} = \text{privateState} \]
2.4 Specification of the Memoir-Basic System

MODULE MemoirLL1Specification

This module defines the low-level specification of Memoir-Basic.

There are eight actions:
- LL1MakeInputAvailable
- LL1PerformOperation
- LL1RepeatOperation
- LL1Restart
- LL1ReadDisk
- LL1WriteDisk
- LL1CorruptRAM
- LL1RestrictedCorruption

EXTENDS MemoirLLPrimitives

The disk and RAM are untrusted. They each can store the service’s public state, encrypted private state, a history summary (a chained hash of all inputs that the service has processed), and an authenticator (a MAC that binds the history summary to the public and private state).

\[ LL1UntrustedStorageType \triangleq [ \]
\[ \text{publicState} : \text{PublicStateType}, \]
\[ \text{privateStateEnc} : \text{PrivateStateEncType}, \]
\[ \text{historySummary} : \text{HashType}, \]
\[ \text{authenticator} : \text{MACType} ] \]

The NVRAM is trusted, since the TPM guarantees that it can only be read or written by the code that implements Memoir. The NVRAM stores the current history summary and the symmetric key that is used (1) to encrypt the private state and (2) to MAC the history summary and service state into an authenticator.

\[ LL1TrustedStorageType \triangleq [ \]
\[ \text{historySummary} : \text{HashType}, \]
\[ \text{symmetricKey} : \text{SymmetricKeyType} ] \]

\[ LL1AvailableInputs \] and \[ LL1ObservedOutputs \] are abstract variables that do not directly represent part of the implementation. They correspond to the HLAvailableInputs and HLObservedOutputs variables in the high-level spec.

\[ \text{VARIABLE LL1AvailableInputs} \]
\[ \text{VARIABLE LL1ObservedOutputs} \]

The \[ LL1ObservedAuthenticators \] variable is also abstract. It records the set of authenticators that the user has seen from Memoir. A malicious user can attempt to use these state authenticators in a replay attack against Memoir.

\[ \text{VARIABLE LL1ObservedAuthenticators} \]

The \[ LL1Disk, LL1RAM, \] and \[ LL1NVRAM \] variables represent concrete state maintained by the Memoir-Basic implementation.

\[ \text{VARIABLE LL1Disk} \]
\[ \text{VARIABLE LL1RAM} \]
\[ \text{VARIABLE LL1NVRAM} \]

\[ LL1TypeInvariant \triangleq \]
\[ \land LL1AvailableInputs \subseteq \text{InputType} \]
\[ \land LL1ObservedOutputs \subseteq \text{OutputType} \]
\[ \land LL1ObservedAuthenticators \subseteq \text{MACType} \]
\( \land LL1Disk \in LL1UntrustedStorageType \)
\( \land LL1RAM \in LL1UntrustedStorageType \)
\( \land LL1NVRAM \in LL1TrustedStorageType \)

\[
LL1Vars \triangleq \langle \\
LL1AvailableInputs, \\
LL1ObservedOutputs, \\
LL1ObservedAuthenticators, \\
LL1Disk, \\
LL1RAM, \\
LL1NVRAM \rangle
\]

The \( LL1MakeInputAvailable \) action is a direct analog of the \( HLMakeInputAvailable \) action in the high-level spec.

\[
LL1MakeInputAvailable \triangleq \\
\exists \ input \in InputType : \\
\land \ input \notin LL1AvailableInputs \\
\land LL1AvailableInputs' = LL1AvailableInputs \cup \{ \text{input} \} \\
\land \text{UNCHANGED LL1Disk} \\
\land \text{UNCHANGED LL1RAM} \\
\land \text{UNCHANGED LL1NVRAM} \\
\land \text{UNCHANGED LL1ObservedOutputs} \\
\land \text{UNCHANGED LL1ObservedAuthenticators}
\]

The \( LL1PerformOperation \) action is invoked by the user to perform a service operation. It is intended to provide the semantics of the \( HLAdvanceService \) action in the high-level spec.

\[
LL1PerformOperation \triangleq \\
\exists \ input \in LL1AvailableInputs : \\
\text{LET} \\
\hspace{1cm} \text{stateHash} \triangleq \text{Hash}(LL1RAM.publicState, LL1RAM.privateStateEnc) \\
\hspace{1cm} \text{historyStateBinding} \triangleq \text{Hash}(LL1RAM.historySummary, \text{stateHash}) \\
\hspace{1cm} \text{privateState} \triangleq \text{SymmetricDecrypt}(LL1NVRAM.symmetricKey, LL1RAM.privateStateEnc) \\
\hspace{1cm} \text{sResult} \triangleq \text{Service}(LL1RAM.publicState, \text{privateState}, \text{input}) \\
\hspace{1cm} \text{newPrivateStateEnc} \triangleq \\
\hspace{2cm} \text{SymmetricEncrypt}(LL1NVRAM.symmetricKey, \text{sResult.newPrivateState}) \\
\hspace{1cm} \text{newHistorySummary} \triangleq \text{Hash}(LL1NVRAM.historySummary, \text{input}) \\
\hspace{1cm} \text{newStateHash} \triangleq \text{Hash}(\text{sResult.newPublicState}, \text{newPrivateStateEnc}) \\
\hspace{1cm} \text{newHistoryStateBinding} \triangleq \text{Hash}(\text{newHistorySummary}, \text{newStateHash}) \\
\hspace{1cm} \text{newAuthenticator} \triangleq \text{GenerateMAC}(LL1NVRAM.symmetricKey, \text{newHistoryStateBinding})
\]

\text{IN}

There are two enablement conditions: First, the authenticator supplied by the user (\( LL1RAM.authenticator \)) must validly bind the user-supplied public and encrypted private state to the history summary supplied by the user. Second, the history summary supplied by the user must match the history summary in the \( NVRAM \).

\[
\land \text{ValidateMAC}(LL1NVRAM.symmetricKey, \text{historyStateBinding}, LL1RAM.authenticator) \\
\land LL1NVRAM.historySummary = LL1RAM.historySummary \\
\text{At the conclusion of the action, the RAM contains the new public and encrypted private state, the new history summary, and an authenticator that binds these together.}
\]

\( \land LL1RAM' = [ \)
publicState ↦ sResult.newPublicState,
privateStateEnc ↦ newPrivateStateEnc,
historySummary ↦ newHistorySummary,
authenticator ↦ newAuthenticator

The NVRAM is updated with the new history summary.
∧ LL1NVRAM’ = [
  historySummary ↦ newHistorySummary,
  symmetricKey ↦ LL1NVRAM.symmetricKey]

The output of the service is added to the set of outputs that the user has observed.
∧ LL1ObservedOutputs’ = LL1ObservedOutputs ∪ {sResult.output}

The disk is unchanged.
∧ UNCHANGED LL1Disk

The set of available inputs is unchanged
∧ UNCHANGED LL1AvailableInputs

The new authenticator is added to the set of authenticators that the user has observed.
∧ LL1ObservedAuthenticators’ =
  LL1ObservedAuthenticators ∪ {newAuthenticator}

The LL1RepeatOperation action is invoked by the user when the computer crashed after a LL1PerformOperation action was performed but before the user had a chance to perform a LL1WriteDisk action to persistently record the new state and its authenticator. This action enables the user to reproduce the result of the most-recent LL1PerformOperation action.

LL1RepeatOperation ≜
  ∃ input ∈ LL1AvailableInputs :
    LET
      stateHash ≜ Hash(LL1RAM.publicState, LL1RAM.privateStateEnc)
      historyStateBinding ≜ Hash(LL1RAM.historySummary, stateHash)
      privateState ≜ SymmetricDecrypt(LL1NVRAM.symmetricKey, LL1RAM.privateStateEnc)
      sResult ≜ Service(LL1RAM.publicState, privateState, input)
      newPrivateStateEnc ≜
        SymmetricEncrypt(LL1NVRAM.symmetricKey, sResult.newPrivateState)
      newStateHash ≜ Hash(sResult.newPublicState, newPrivateStateEnc)
      newHistoryStateBinding ≜ Hash(LL1NVRAM.historySummary, newStateHash)
      newAuthenticator ≜ GenerateMAC(LL1NVRAM.symmetricKey, newHistoryStateBinding)
    IN
    There are two enablement conditions: First, the authenticator supplied by the user (LL1RAM.authenticator) must validly bind the user-supplied public and encrypted private state to the history summary supplied by the user.
    Second, the history summary supplied by the user, hashed with the input supplied by the user, must match the history summary in the NVRAM. This condition ensures that this action will invoke the service with the same input used with the most recent LL1PerformOperation action.
    ∧ ValidateMAC(LL1NVRAM.symmetricKey, historyStateBinding, LL1RAM.authenticator)
    ∧ LL1NVRAM.historySummary = Hash(LL1RAM.historySummary, input)
    At the conclusion of the action, the RAM contains the new public and encrypted private state, the new history summary, and an authenticator that binds these together. These should match the values produced by the most recent LL1PerformOperation action.
    ∧ LL1RAM’ = [
      publicState ↦ sResult.newPublicState,
The output of the service is added to the set of outputs that the user has observed. If Memoir is working correctly, the user already saw this output when the previous \(LL1\text{PerformOperation}\) action was executed.

\[
\land LL1\text{ObservedOutputs}' = LL1\text{ObservedOutputs} \cup \{sResult.output\}
\]

The NVRAM is unchanged, because this action is not supposed to change the state of the service.

\[
\land \text{UNCHANGED } LL1\text{NVRAM}
\]

The disk is unchanged.

\[
\land \text{UNCHANGED } LL1\text{Disk}
\]

The set of available inputs is unchanged

\[
\land \text{UNCHANGED } LL1\text{AvailableInputs}
\]

The new authenticator is added to the set of authenticators that the user has observed. If Memoir is working correctly, the user already saw this authenticator when the previous \(LL1\text{PerformOperation}\) action was executed.

\[
\land LL1\text{ObservedAuthenticators}' = LL1\text{ObservedAuthenticators} \cup \{\text{newAuthenticator}\}
\]

The \(LL1\text{Restart}\) action occurs when the computer restarts.

\[
LL1\text{Restart} \triangleq \exists \text{untrustedStorage} \in LL1\text{UntrustedStorageType}, \text{randomSymmetricKey} \in \text{SymmetricKeyType} \setminus \{LL1\text{NVRAM}.\text{symmetricKey}\}, \text{hash} \in \text{HashType}:
\]

The state of the RAM is trashed by a restart, so we set it to some *almost* arbitrary value in \(LL1\text{UntrustedStorageType}\). The only condition we impose is that the authenticator is not coincidentally equal to an authenticator that could be computed with the symmetric key known only to Memoir.

\[
\land \text{untrustedStorage}.\text{authenticator} = \text{GenerateMAC}(\text{randomSymmetricKey}, \text{hash})
\land LL1\text{RAM}' = \text{untrustedStorage}
\land \text{UNCHANGED } LL1\text{Disk}
\land \text{UNCHANGED } LL1\text{NVRAM}
\land \text{UNCHANGED } LL1\text{AvailableInputs}
\land \text{UNCHANGED } LL1\text{ObservedOutputs}
\land \text{UNCHANGED } LL1\text{ObservedAuthenticators}
\]

The \(LL1\text{ReadDisk}\) action copies the state of the disk into the RAM.

\[
LL1\text{ReadDisk} \triangleq
\land LL1\text{RAM}' = LL1\text{Disk}
\land \text{UNCHANGED } LL1\text{Disk}
\land \text{UNCHANGED } LL1\text{NVRAM}
\land \text{UNCHANGED } LL1\text{AvailableInputs}
\land \text{UNCHANGED } LL1\text{ObservedOutputs}
\land \text{UNCHANGED } LL1\text{ObservedAuthenticators}
\]

The \(LL1\text{WriteDisk}\) action copies the state of the RAM onto the disk.
The $LL_1$ \texttt{WriteDisk} action models the ability of a malicious user to attack Memoir by supplying \textit{almost} arbitrary data to Memoir. The data is not completely arbitrary, because the user is assumed to be unable to forge authenticators using the symmetric key stored in the \texttt{NVRAM} of Memoir.

\[
LL_1 \texttt{WriteDisk} \triangleq \\
\land LL_1 \texttt{Disk}' = LL_1 \texttt{RAM} \\
\land \text{UNCHANGED } LL_1 \texttt{RAM} \\
\land \text{UNCHANGED } LL_1 \texttt{NVRAM} \\
\land \text{UNCHANGED } LL_1 \texttt{AvailableInputs} \\
\land \text{UNCHANGED } LL_1 \texttt{ObservedOutputs} \\
\land \text{UNCHANGED } LL_1 \texttt{ObservedAuthenticators}
\]

The $LL_1$ \texttt{CorruptRAM} action models the ability of a malicious user to attack Memoir by supplying \textit{almost} arbitrary data to Memoir. The data is not completely arbitrary, because the user is assumed to be unable to forge authenticators using the symmetric key stored in the \texttt{NVRAM} of Memoir.

\[
LL_1 \texttt{CorruptRAM} \triangleq \\
\exists \text{untrustedStorage} \in LL_1 \texttt{UntrustedStorageType}, \\
fake\text{SymmetricKey} \in \text{SymmetricKeyType} \setminus \{LL_1 \texttt{NVRAM}.\text{symmetricKey}\}, \\
\text{hash} \in \text{HashType} : \\
\land \lor \text{untrustedStorage}.\text{authenticator} \in LL_1 \texttt{ObservedAuthenticators} \\
\lor \text{The user can launch a replay attack by re-using any authenticator previously observed.} \\
\land \lor \text{untrustedStorage}.\text{authenticator} = \text{GenerateMAC}(\text{fake\text{SymmetricKey}}, \text{hash}) \\
\land \land LL_1 \texttt{RAM}' = \text{untrustedStorage} \\
\land \text{UNCHANGED } LL_1 \texttt{Disk} \\
\land \text{UNCHANGED } LL_1 \texttt{NVRAM} \\
\land \text{UNCHANGED } LL_1 \texttt{AvailableInputs} \\
\land \text{UNCHANGED } LL_1 \texttt{ObservedOutputs} \\
\land \text{UNCHANGED } LL_1 \texttt{ObservedAuthenticators}
\]

The $LL_1$ \texttt{CorruptRAM} action does not model any realistic action in a direct implementation of this low-level spec. The TPM prevents any code other than Memoir from writing to the \texttt{NVRAM}, so an attacker cannot actually perform this action in Memoir-Basic.

However, Memoir-Opt allows an attacker to perform an action ($LL_2$\texttt{CorruptSPCR}) that corrupts the stored history summary in Memoir-Opt, and in the correctness proof of Memoir-Opt, we will show that the \texttt{CorruptSPCR} action in Memoir-Opt (under some circumstances) refines to the $LL_1$\texttt{RestrictedCorruption} action in Memoir-Basic.

Similarly, when a $LL_2$\texttt{Restart} action occurs in Memoir-Opt, there are circumstances that cause this to refine to the $LL_1$\texttt{RestrictedCorruption} action in Memoir-Basic.

For this reason, we need the $LL_1$\texttt{RestrictedCorruption} action to be strong enough to enable refinement from the $LL_2$\texttt{CorruptSPCR} and $LL_2$\texttt{Restart} actions in Memoir-Opt, but weak enough to enable refinement to an action in the high-level spec, specifically the \texttt{HLDie} action.

We therefore impose a somewhat bizarre-looking pair of constraints on the garbage value to which an attacker can set the history summary in the \texttt{NVRAM}. The first constraint (labeled \textit{current}) is needed to ensure that the \texttt{CardinalityInvariant} and \texttt{UniquenessInvariant} continue to hold when a $LL_1$\texttt{RestrictedCorruption} action occurs, and the second constraint (labeled \textit{previous}) is needed to ensure that the \texttt{InclusionInvariant} continues to hold when a $LL_1$\texttt{RestrictedCorruption} action occurs.

\[
LL_1 \texttt{RestrictedCorruption} \triangleq \\
\land \land \text{nvram}:: \\
\land \exists \text{garbageHistorySummary} \in \text{HashType} : \\
\land \land \text{current}(\text{garbageHistorySummary})::
\]
There is no authenticator that validates a history state binding that binds the garbage history summary to any state hash.

∀ stateHash ∈ HashType, authenticator ∈ LL1ObservedAuthenticators :

LET

historyStateBinding ≝ Hash(garbageHistorySummary, stateHash)

IN

¬ValidateMAC(LL1NVRAM.symmetricKey, historyStateBinding, authenticator)

∧ previous(garbageHistorySummary):=:

There is no authenticator that validates a history state binding that binds any predecessor of the garbage history summary to any state hash.

∀ stateHash ∈ HashType, authenticator ∈ LL1ObservedAuthenticators, someHistorySummary ∈ HashType, someInput ∈ InputType :

LET

historyStateBinding ≝ Hash(someHistorySummary, stateHash)

IN

garbageHistorySummary = Hash(someHistorySummary, someInput) ⇒

¬ValidateMAC(LL1NVRAM.symmetricKey, historyStateBinding, authenticator)

The history summary in the NVRAM becomes equal to the garbage history summary.

∧ LL1NVRAM’ = [ historySummary ↦→ garbageHistorySummary, symmetricKey ↦→ LL1NVRAM.symmetricKey ]

∧ ram::

∧ unchanged::

A LL2CorruptSPCR action in the Memoir-Opt spec leaves the state of the RAM unchanged.

UNCHANGED LL1RAM

∧ trashed::

A LL2Restart action in the Memoir-Opt spec trashes the RAM, so we set it to some *almost* arbitrary value in LL1UntrustedStorageType. The only condition we impose is that the authenticator is not coincidentally equal to an authenticator that could be computed with the symmetric key known only to Memoir.

∃ untrustedStorage ∈ LL1UntrustedStorageType, randomSymmetricKey ∈ SymmetricKeyType \ { LL1NVRAM.symmetricKey }, hash ∈ HashType :

∧ untrustedStorage.authenticator = GenerateMAC(randomSymmetricKey, hash)

∧ LL1RAM’ = untrustedStorage

∧ UNCHANGED LL1Disk

∧ UNCHANGED LL1AvailableInputs

∧ UNCHANGED LL1ObservedOutputs

∧ UNCHANGED LL1ObservedAuthenticators

In the initial state of the Memoir-Basic implementation, some symmetric key is generated and stored in the NVRAM. The initial history summary is the base hash value, indicating that no inputs have been supplied yet. An initial authenticator binds the initial history summary to the initial public and encrypted private state.

LL1Init ≝

∃ symmetricKey ∈ SymmetricKeyType :

LET

initialPrivateStateEnc ≝ SymmetricEncrypt(symmetricKey, InitialPrivateState)

initialStateHash ≝ Hash(InitialPublicState, initialPrivateStateEnc)
\[\text{initialHistoryStateBinding} \triangleq \text{Hash}(\text{BaseHashValue}, \text{initialStateHash})\]
\[\text{initialAuthenticator} \triangleq \text{GenerateMAC}(\text{symmetricKey}, \text{initialHistoryStateBinding})\]
\[\text{initialUntrustedStorage} \triangleq [\]
\hspace{1em} \text{publicState} \mapsto \text{InitialPublicState},
\hspace{1em} \text{privateStateEnc} \mapsto \text{initialPrivateStateEnc},
\hspace{1em} \text{historySummary} \mapsto \text{BaseHashValue},
\hspace{1em} \text{authenticator} \mapsto \text{initialAuthenticator}\]
\[\text{initialTrustedStorage} \triangleq [\]
\hspace{1em} \text{historySummary} \mapsto \text{BaseHashValue},
\hspace{1em} \text{symmetricKey} \mapsto \text{symmetricKey}\]
\[
\text{in} \wedge \text{LL}_1\text{Disk} = \text{initialUntrustedStorage} \\
\wedge \text{LL}_1\text{RAM} = \text{initialUntrustedStorage} \\
\wedge \text{LL}_1\text{NVRAM} = \text{initialTrustedStorage} \\
\wedge \text{LL}_1\text{AvailableInputs} = \text{InitialAvailableInputs} \\
\wedge \text{LL}_1\text{ObservedOutputs} = \{\} \\
\wedge \text{LL}_1\text{ObservedAuthenticators} = \{\text{initialAuthenticator}\}
\]
\[\text{LL}_1\text{Next} \triangleq \]
\[\checkmark \text{LL}_1\text{MakeInputAvailable} \\
\checkmark \text{LL}_1\text{PerformOperation} \\
\checkmark \text{LL}_1\text{RepeatOperation} \\
\checkmark \text{LL}_1\text{Restart} \\
\checkmark \text{LL}_1\text{ReadDisk} \\
\checkmark \text{LL}_1\text{WriteDisk} \\
\checkmark \text{LL}_1\text{CorruptRAM} \\
\checkmark \text{LL}_1\text{RestrictedCorruption}
\]
\[\text{LL}_1\text{Spec} \triangleq \text{LL}_1\text{Init} \wedge \Box[\text{LL}_1\text{Next}]_{LL_1\text{Vars}}\]
This module defines the specification of Memoir-Opt.

There are nine actions:

- LL2MakeInputAvailable
- LL2PerformOperation
- LL2RepeatOperation
- LL2TakeCheckpoint
- LL2Restart
- LL2ReadDisk
- LL2WriteDisk
- LL2CorruptRAM
- LL2CorruptSPCR

**EXTENDS** MemoirLL1Implementation

In Memoir-Opt, each history summary (which is a composite chained hash of all inputs that the service has processed) is partitioned into two pieces: an anchor and an extension.

\[
\text{HistorySummaryType} \triangleq [ \\
\text{anchor} : \text{HashType} , \\
\text{extension} : \text{HashType} ]
\]

The disk and RAM are untrusted. They each can store the service’s public state, encrypted private state, a history summary, and an authenticator (a MAC that binds the history summary to the public and private state).

\[
\text{LL2UntrustedStorageType} \triangleq [ \\
\text{publicState} : \text{PublicStateType} , \\
\text{privateStateEnc} : \text{PrivateStateEncType} , \\
\text{historySummary} : \text{HistorySummaryType} , \\
\text{authenticator} : \text{MACType} ]
\]

The NVRAM is trusted, since the TPM guarantees that it can only be read or written by the code that implements Memoir. The NVRAM stores the current history summary anchor and the symmetric key that is used to encrypt the private state and to MAC the history summary and service state into an authenticator. It also stores a hash barrier for securing the history summary and a guard bit that indicates whether the current history summary has been extended, such that the history summary anchor is not a complete representation of the inputs summary.

\[
\text{LL2TrustedStorageType} \triangleq [ \\
\text{historySummaryAnchor} : \text{HashType} , \\
\text{symmetricKey} : \text{SymmetricKeyType} , \\
\text{hashBarrier} : \text{HashType} , \\
\text{extensionInProgress} : \text{BOOLEAN} ]
\]

**LL2AvailableInputs** and **LL2ObservedOutputs** are abstract variables that do not directly represent part of the implementation. They correspond to the **HLAvailableInputs** and **HLObservedOutputs** variables in the Memoir-Opt spec.

**VARIABLE** LL2AvailableInputs

**VARIABLE** LL2ObservedOutputs

The **ObservedAuthenticators** variable is also abstract. It records the set of authenticators that the user has seen from Memoir. A malicious user can attempt to use these authenticators in a replay attack against Memoir.

**VARIABLE** LL2ObservedAuthenticators
The $LL2\text{Disk}$, $LL2\text{RAM}$, $LL2\text{NVRAM}$, and $LL2\text{SPCR}$ variables represent concrete state maintained by the Memoir-Opt implementation.

The $LL2\text{SPCR}$ is semi-trusted. Any party can write to it, but arbitrary writes are not allowed. The only allowable updates are of the form

$$\exists x : LL2\text{SPCR}' = \text{Hash}(LL2\text{SPCR}, x)$$

We use the $LL2\text{SPCR}$ to store a history summary extension.

\begin{verbatim}
VARIABLE LL2Disk
VARIABLE LL2RAM
VARIABLE LL2NVRAM
VARIABLE LL2SPCR

LL2TypeInvariant ≜
∧ LL2AvailableInputs ⊆ InputType
∧ LL2ObservedOutputs ⊆ OutputType
∧ LL2ObservedAuthenticators ⊆ MACType
∧ LL2Disk ∈ LL2UntrustedStorageType
∧ LL2RAM ∈ LL2UntrustedStorageType
∧ LL2NVRAM ∈ LL2TrustedStorageType
∧ LL2SPCR ∈ HashType

LL2Vars ≜ (
LL2AvailableInputs,
LL2ObservedOutputs,
LL2ObservedAuthenticators,
LL2Disk,
LL2RAM,
LL2NVRAM,
LL2SPCR)
\end{verbatim}

The $\text{Checkpoint}$ function takes a history summary that may or may not be checkpointed and produces a checkpointed history summary from it.

\begin{verbatim}
Checkpoint(historySummary) ≜
LET
checkpointedAnchor ≜ \text{Hash}(historySummary.\text{anchor}, historySummary.\text{extension})
checkpointedHistorySummary ≜ [
  anchor → checkpointedAnchor,
  extension → BaseHashValue]
IN
IF historySummary.\text{extension} = BaseHashValue
THEN
historySummary
ELSE
checkpointedHistorySummary
\end{verbatim}

The $\text{Successor}$ function defines the history summary that results from extending a given history summary with a given input. It secures the input using a hash barrier to thwart forgery.

\begin{verbatim}
Successor(historySummary, input, hashBarrier) ≜
LET
\end{verbatim}
securedInput $\triangleq$ Hash(hashBarrier, input)
newAnchor $\triangleq$ historySummary.anchor
newExtension $\triangleq$ Hash(historySummary.extension, securedInput)
newHistorySummary $\triangleq$ [
  anchor $\mapsto$ newAnchor,
  extension $\mapsto$ newExtension
] IN newHistorySummary

The L2MakeInputAvailable action is a direct analog of the HLMakeInputAvailable action in the high-level spec.

$L2\text{MakeInputAvailable} \triangleq$
$\exists$ input $\in$ InputType :
  $\land$ input $\notin$ L2AvailableInputs
  $\land$ L2AvailableInputs' = L2AvailableInputs $\cup$ {input}
  $\land$ UNCHANGED L2Disk
  $\land$ UNCHANGED L2RAM
  $\land$ UNCHANGED L2NVRAM
  $\land$ UNCHANGED L2SPCR
  $\land$ UNCHANGED L2ObservedOutputs
  $\land$ UNCHANGED L2ObservedAuthenticators

The $L2\text{PerformOperation}$ action is invoked by the user to perform a service operation. It is intended to provide the semantics of the HLAdvanceService action in the high-level spec.

$L2\text{PerformOperation} \triangleq$
$\exists$ input $\in$ L2AvailableInputs :
  LET
    historySummaryHash $\triangleq$  
      Hash(L2RAM.historySummary.anchor, L2RAM.historySummary.extension)
    stateHash $\triangleq$  
      Hash(L2RAM.publicState, L2RAM.privateStateEnc)
    historyStateBinding $\triangleq$  
      Hash(historySummaryHash, stateHash)
    privateState $\triangleq$  
      SymmetricDecrypt(L2NVRAM.symmetricKey, L2RAM.privateStateEnc)
    sResult $\triangleq$  
      Service(L2RAM.publicState, privateState, input)
    newPrivateStateEnc $\triangleq$  
      SymmetricEncrypt(L2NVRAM.symmetricKey, sResult.newPrivateState)
    currentHistorySummary $\triangleq$  
      [ anchor $\mapsto$ L2NVRAM.historySummaryAnchor, extension $\mapsto$ L2SPCR]
    newHistorySummary $\triangleq$  
      Successor(currentHistorySummary, input, L2NVRAM.hashBarrier)
    newHistorySummaryHash $\triangleq$  
      Hash(newHistorySummary.anchor, newHistorySummary.extension)
    newStateHash $\triangleq$  
      Hash(sResult.newPublicState, newPrivateStateEnc)
    newHistoryStateBinding $\triangleq$  
      Hash(newHistorySummaryHash, newStateHash)
    newAuthenticator $\triangleq$  
      GenerateMAC(L2NVRAM.symmetricKey, newHistoryStateBinding)

IN

There are three enablement conditions:

First, the authenticator supplied by the user (L2RAM.authenticator) must validly bind the user-supplied public and encrypted private state to the user-supplied history summary.
Second, the value in the SPCR must be consistent with the flag that indicates whether an extension is in progress.

Third, the user-supplied history summary (LL2RAM.historySummary) must match the history summary in the TPM (NVRAM and SPCR). There are two different ways to check this condition, depending on whether a extension is in progress.

\[\text{ValidateMAC}(LL2NVRAM.symmetricKey, \text{historyStateBinding}, LL2RAM.authenticator)\]
\[\text{IF } LL2NVRAM.extensionInProgress = \text{TRUE} \text{ THEN} \]
\[\text{∧ } LL2SPCR \neq \text{BaseHashValue} \]
\[\text{∧ } \text{currentHistorySummary} = LL2RAM.historySummary\]
\[\text{ELSE} \]
\[\text{∧ } LL2SPCR = \text{BaseHashValue} \]
\[\text{∧ } \text{currentHistorySummary} = \text{Checkpoint}(LL2RAM.historySummary)\]

At the conclusion of the action, the RAM contains the new public and encrypted private state, the new history summary, and an authenticator that binds these together.

\[LL2RAM' = [\]
\[\text{publicState} \mapsto \text{sResult.newPublicState}, \]
\[\text{privateStateEnc} \mapsto \text{newPrivateStateEnc}, \]
\[\text{historySummary} \mapsto \text{newHistorySummary}, \]
\[\text{authenticator} \mapsto \text{newAuthenticator}]\]

The NVRAM is updated to indicate that the extension is in progress. The NVRAM may already indicate this, in which case this step does not require writing to the NVRAM.

\[LL2NVRAM' = [\]
\[\text{historySummaryAnchor} \mapsto LL2NVRAM.historySummaryAnchor, \]
\[\text{symmetricKey} \mapsto LL2NVRAM.symmetricKey, \]
\[\text{hashBarrier} \mapsto LL2NVRAM.hashBarrier, \]
\[\text{extensionInProgress} \mapsto \text{TRUE}]\]

The SPCR is updated with the new history summary extension.

\[LL2SPCR' = \text{newHistorySummary.extension}\]

The output of the service is added to the set of outputs that the user has observed.

\[LL2ObservedOutputs' = LL2ObservedOutputs \cup \{\text{sResult.output}\}\]

The disk is unchanged.

\[\text{UNCHANGED } LL2Disk\]

The set of available inputs is unchanged.

\[\text{UNCHANGED } LL2AvailableInputs\]

The new authenticator is added to the set of authenticators that the user has observed.

\[LL2ObservedAuthenticators' = LL2ObservedAuthenticators \cup \{\text{newAuthenticator}\}\]

The LL2RepeatOperation action is invoked by the user when the computer crashed after a LL2PerformOperation action was performed but before the user had a chance to perform a LL2WriteDisk action to persistently record the new state and its authenticator. This action enables the user to reproduce the result of the most-recent LL2PerformOperation action.

\[LL2RepeatOperation \triangleq \]
\[\exists \text{input} \in LL2AvailableInputs : \]
\[\text{LET}\]
\[\text{historySummaryHash} \triangleq \]
\[\text{Hash}(LL2RAM.historySummary.\text{anchor}, LL2RAM.historySummary.\text{extension})\]
\[
\text{stateHash} \triangleq \text{Hash}(LL2RAM\.publicState,(LL2RAM\.privateStateEnc))
\]
\[
\text{historyStateBinding} \triangleq \text{Hash}((\text{historySummary}Hash,\text{stateHash}))
\]
\[
\text{newCheckpointedHistorySummary} \triangleq \text{Successor}(LL2RAM\.historySummary, \text{input},LL2NVRAM\.hashBarrier)
\]
\[
\text{checkpointedNewCheckpointedHistorySummary} \triangleq \text{Checkpoint(newHistorySummary)}
\]
\[
\text{privateState} \triangleq \text{SymmetricDecrypt}(LL2NVRAM\.symmetricKey,LL2RAM\.privateStateEnc)
\]
\[
\text{sResult} \triangleq \text{Service}(LL2RAM\.publicState,\text{privateState},\text{input})
\]
\[
\text{newPrivateStateEnc} \triangleq \text{SymmetricEncrypt}(LL2NVRAM\.symmetricKey,sResult\.newPrivateState)
\]
\[
\text{currentHistorySummary} \triangleq \,[
\begin{align*}
\text{anchor} &\mapsto LL2NVRAM\.historySummaryAnchor, \\
\text{extension} &\mapsto LL2SPCR
\end{align*}
\]
\[
\text{currentHistorySummaryHash} \triangleq \text{Hash}(\text{currentHistorySummaryAnchor},LL2SPCR)
\]
\[
\text{newStateHash} \triangleq \text{Hash}(\text{sResult\.newPublicState,},\text{newPrivateStateEnc})
\]
\[
\text{newHistoryStateBinding} \triangleq \text{Hash}(\text{currentHistorySummaryHash,},\text{newStateHash})
\]
\[
\text{newAuthenticator} \triangleq \text{GenerateMAC}(LL2NVRAM\.symmetricKey,\text{newHistoryStateBinding})
\]

There are three enablement conditions:

First, the authenticator supplied by the user (\text{LL2RAM\.authenticator}) must validly bind the user-supplied public and encrypted private state to the history summary supplied by the user.

Second, a \text{TakeCheckpoint} action should always occur immediately before a shutdown, power-off, or reboot; therefore, an extension will not be in progress at the time \text{LL2RepeatOperation} is needed. This implies that the flag in the NVRAM must indicate that an extension is not in progress, and the value in the SPCR must equal the \text{BaseHashValue}.

Third, the user-supplied history summary (\text{LL2RAM\.historySummary}), extended with the user-supplied input, must match the history summary in the TPM (NVRAM and SPCR). This condition ensures that this action will invoke the service with the same input used with the most recent \text{LL2PerformOperation} action. There are two different ways to check this condition, depending on whether a checkpoint was taken before the input was processed.

\[\wedge \text{ValidateMAC}(LL2NVRAM\.symmetricKey,\text{historyStateBinding, LL2RAM\.authenticator})\]
\[\wedge \text{LL2NVRAM\.extensionInProgress} = \text{FALSE}\]
\[\wedge \text{LL2SPCR} = \text{BaseHashValue}\]
\[\wedge \begin{align*}
\text{no checkpoint before input} \\
\text{currentHistorySummary} &= \text{checkpointedNewHistorySummary} \\
\text{checkpoints before input} \\
\text{currentHistorySummary} &= \text{checkpointedNewCheckpointedHistorySummary}
\end{align*}\]

At the conclusion of the action, the RAM contains the new public and encrypted private state, the new history summary, and an authenticator that binds these together. These should match the values produced by the most recent \text{LL2PerformOperation} action.

\[\wedge \text{LL2RAM}' = [
\begin{align*}
\text{publicState} &\mapsto \text{sResult\.newPublicState,} \\
\text{privateStateEnc} &\mapsto \text{newPrivateStateEnc,} \\
\text{historySummary} &\mapsto \text{currentHistorySummary,} \\
\text{authenticator} &\mapsto \text{newAuthenticator}
\end{align*}
\]

The output of the service is added to the set of outputs that the user has observed. If Memoir is working correctly, the user already saw this output when the previous \text{LL2PerformOperation} action was executed.

\[\wedge \text{LL2ObservedOutputs'} = \text{LL2ObservedOutputs} \cup \{\text{sResult.output}\}\]
The NVRAM is unchanged, because this action is not supposed to change the state of the service.
∧ UNCHANGED LL2NVRAM
The SPCR is unchanged, because this action is not supposed to change the state of the service.
∧ UNCHANGED LL2SPCR
The disk is unchanged.
∧ UNCHANGED LL2Disk
The set of available inputs is unchanged.
∧ UNCHANGED LL2AvailableInputs
The new authenticator is added to the set of authenticators that the user has observed. If Memoir is working correctly, the user already saw this authenticator when the previous LL2PerformOperation action was executed.
∧ LL2ObservedAuthenticators' = LL2ObservedAuthenticators ∪ \{newAuthenticator\}

The LL2TakeCheckpoint action occurs in response to an NMI indicating that a shutdown, power-off, or reboot is imminent.

\[
\text{LL2TakeCheckpoint } \triangleq \text{LET} \\
\text{newHistorySummaryAnchor } \triangleq \text{Hash}(\text{LL2NVRAM.historySummaryAnchor, LL2SPCR}) \\
\text{IN} \\
\text{There are two enableness conditions. The guard bit in the NVRAM must indicate that an extension is in progress, and the SPCR must contain an extension.} \\
∧ \text{LL2NVRAM.extensionInProgress = TRUE} \\
∧ \text{LL2SPCR} \neq \text{BaseHashValue} \\
\text{This action changes nothing other than the NVRAM.} \\
∧ \text{UNCHANGED LL2RAM} \\
∧ \text{UNCHANGED LL2Disk} \\
∧ \text{LL2NVRAM'} = [ \\
\text{historySummaryAnchor } \mapsto \text{newHistorySummaryAnchor}, \\
\text{symmetricKey } \mapsto \text{LL2NVRAM.symmetricKey}, \\
\text{hashBarrier } \mapsto \text{LL2NVRAM.hashBarrier}, \\
\text{extensionInProgress } \mapsto \text{FALSE}] \\
∧ \text{UNCHANGED LL2SPCR} \\
∧ \text{UNCHANGED LL2AvailableInputs} \\
∧ \text{UNCHANGED LL2ObservedOutputs} \\
∧ \text{UNCHANGED LL2ObservedAuthenticators} \\
\]

The LL2Restart action occurs when the computer restarts.

\[
\text{LL2Restart } \triangleq \exists \text{untrustedStorage } \in \text{LL2UntrustedStorageType}, \\
\text{randomSymmetricKey } \in \text{SymmetricKeyType} \setminus \{\text{LL2NVRAM.symmetricKey}\}, \\
\text{hash } \in \text{HashType} : \\
\text{The state of the RAM is garbaged by a restart, so we set it to some *almost* arbitrary value in LL2UntrustedStorageType. The only condition we impose is that the authenticator is not coincidentally equal to an authenticator that could be computed with the symmetric key known only to Memoir.} \\
∧ \text{untrustedStorage.authenticator } = \text{GenerateMAC(randomSymmetricKey, hash)} \\
∧ \text{LL2RAM'} = \text{untrustedStorage} \\
\]

36
\[ LL_2\text{Disk} \]
\[ LL_2\text{NVRAM} \]
\[ LL_2\text{SPCR} \]
\[ LL_2\text{AvailableInputs} \]
\[ LL_2\text{ObservedOutputs} \]
\[ LL_2\text{ObservedAuthenticators} \]

The value of the \( SPCR \) is set to a known starting value, which we model with the \( BaseHashValue \).

\[ LL_2\text{SPCR}' = BaseHashValue \]
\[ LL_2\text{AvailableInputs} \]
\[ LL_2\text{ObservedOutputs} \]
\[ LL_2\text{ObservedAuthenticators} \]

The \( LL_2\text{ReadDisk} \) action copies the state of the disk into the RAM.

\[ LL_2\text{ReadDisk} \triangleq \]
\[ \land \ LL_2\text{RAM}' = LL_2\text{Disk} \]
\[ \land \ UNCHANGED \ LL_2\text{Disk} \]
\[ \land \ UNCHANGED \ LL_2\text{NVRAM} \]
\[ \land \ UNCHANGED \ LL_2\text{SPCR} \]
\[ \land \ UNCHANGED \ LL_2\text{AvailableInputs} \]
\[ \land \ UNCHANGED \ LL_2\text{ObservedOutputs} \]
\[ \land \ UNCHANGED \ LL_2\text{ObservedAuthenticators} \]

The \( LL_2\text{WriteDisk} \) action copies the state of the RAM onto the disk.

\[ LL_2\text{WriteDisk} \triangleq \]
\[ \land \ LL_2\text{Disk}' = LL_2\text{RAM} \]
\[ \land \ UNCHANGED \ LL_2\text{RAM} \]
\[ \land \ UNCHANGED \ LL_2\text{NVRAM} \]
\[ \land \ UNCHANGED \ LL_2\text{SPCR} \]
\[ \land \ UNCHANGED \ LL_2\text{AvailableInputs} \]
\[ \land \ UNCHANGED \ LL_2\text{ObservedOutputs} \]
\[ \land \ UNCHANGED \ LL_2\text{ObservedAuthenticators} \]

The \( LL_2\text{CorruptRAM} \) action models the ability of a malicious user to attack Memoir by supplying *almost* arbitrary data to Memoir. The data is not completely arbitrary, because the user is assumed to be unable to forge authenticators using the symmetric key stored in the \text{NVRAM} of the TPM.

\[ LL_2\text{CorruptRAM} \triangleq \]
\[ \exists \ untrustedStorage \in LL_2\text{UntrustedStorageType}, \]
\[ \text{fakeSymmetricKey} \in \text{SymmetricKeyType} \setminus \{\text{LL2NVRAM}.\text{symmetricKey}\}, \]
\[ \text{hash} \in \text{HashType}: \]
\[ \land \ \lor \ untrustedStorage.\text{authenticator} \in LL_2\text{ObservedAuthenticators} \]
\[ \lor \ \text{the user can launch a replay attack by re-using any authenticator previously observed.} \]
\[ \lor \ \text{or the user can create a fake authenticator using some other symmetric key.} \]
\[ \lor \ \text{GenerateMAC(fakeSymmetricKey, hash)} \]
\[ \land \ LL_2\text{RAM}' = untrustedStorage \]
\[ \land \ UNCHANGED \ LL_2\text{Disk} \]
\[ \land \ UNCHANGED \ LL_2\text{NVRAM} \]
\[ \land \ UNCHANGED \ LL_2\text{SPCR} \]
\[ \land \ UNCHANGED \ LL_2\text{AvailableInputs} \]
\[ \land \ UNCHANGED \ LL_2\text{ObservedOutputs} \]

37
The $LL_2\text{CorruptSPCR}$ action models the ability of a malicious user to attack Memoir by extending the $SPCR$ with *almost* arbitrary data. The data is not completely arbitrary, because the user is assumed not to know the hash barrier stored in the $NVRAM$ of the $TPM$, so the user has negligible probability of being able to correctly guess this value and use it to extend the $SPCR$.

$$LL_2\text{CorruptSPCR} \triangleq \exists \text{fakeHash} \in \text{HashDomain} :$$

- The $SPCR$ can only be modified by extending it.
- $new\text{HistorySummaryExtension} \triangleq \text{Hash}(LL_2\text{SPCR}, \text{fakeHash})$

$$\land \forall \text{fakeInput} \in \text{InputType} : \text{fakeHash} \neq \text{Hash}(LL_2\text{NVRAM}.\text{hashBarrier}, \text{fakeInput})$$

$$\land \text{UNCHANGED } LL_2\text{RAM}$$

$$\land \text{UNCHANGED } LL_2\text{Disk}$$

$$\land \text{UNCHANGED } LL_2\text{NVRAM}$$

$$\land LL_2\text{SPCR'} = new\text{HistorySummaryExtension}$$

$$\land \text{UNCHANGED } LL_2\text{AvailableInputs}$$

$$\land \text{UNCHANGED } LL_2\text{ObservedOutputs}$$

$$\land \text{UNCHANGED } LL_2\text{ObservedAuthenticators}$$

In the initial state of the Memoir-Opt spec, some symmetric key and some hash barrier are generated and stored in the $NVRAM$. The initial history summary anchor and extension are both the base hash value, indicating that no inputs have been supplied yet. An initial authenticator binds the initial history summary to the initial public and encrypted private state.

$$LL_2\text{Init} \triangleq$$

- $\exists \text{symmetricKey} \in \text{SymmetricKeyType}, \text{hashBarrier} \in \text{HashType} :$

- Let

- $\text{initialPrivateStateEnc} \triangleq \text{SymmetricEncrypt}(\text{symmetricKey}, \text{InitialPrivateState})$

- $\text{initialStateHash} \triangleq \text{Hash}(\text{InitialPublicState}, \text{initialPrivateStateEnc})$

- $\text{initialHistorySummary} \triangleq [\text{anchor} \mapsto \text{BaseHashValue}, \text{extension} \mapsto \text{BaseHashValue}]$

- $\text{initialHistorySummaryHash} \triangleq \text{Hash}(\text{BaseHashValue}, \text{BaseHashValue})$

- $\text{initialHistoryStateBinding} \triangleq \text{Hash}(\text{initialHistorySummaryHash}, \text{initialStateHash})$

- $\text{initialAuthenticator} \triangleq \text{GenerateMAC}(\text{symmetricKey}, \text{initialHistoryStateBinding})$

- $\text{initialUntrustedStorage} \triangleq [\text{publicState} \mapsto \text{InitialPublicState}, \text{privateStateEnc} \mapsto \text{initialPrivateStateEnc}, \text{historySummary} \mapsto \text{initialHistorySummary}, \text{authenticator} \mapsto \text{initialAuthenticator}]$

- $\text{initialTrustedStorage} \triangleq [\text{historySummaryAnchor} \mapsto \text{BaseHashValue}, \text{symmetricKey} \mapsto \text{symmetricKey}, \text{hashBarrier} \mapsto \text{hashBarrier}, \text{extensionInProgress} \mapsto \text{FALSE}]$

$$\land \text{LL}_2\text{Disk} = \text{initialUntrustedStorage}$$
\[ LL2_{\text{RAM}} = \text{initialUntrustedStorage} \]
\[ LL2_{\text{NVRAM}} = \text{initialTrustedStorage} \]
\[ LL2_{\text{SPCR}} = \text{BaseHashValue} \]
\[ LL2_{\text{AvailableInputs}} = \text{InitialAvailableInputs} \]
\[ LL2_{\text{ObservedOutputs}} = \{\} \]
\[ LL2_{\text{ObservedAuthenticators}} = \{\text{initialAuthenticator}\} \]

\[ LL2_{\text{Next}} \triangleq \]
\[ \lor \ LL2_{\text{MakeInputAvailable}} \]
\[ \lor \ LL2_{\text{PerformOperation}} \]
\[ \lor \ LL2_{\text{RepeatOperation}} \]
\[ \lor \ LL2_{\text{TakeCheckpoint}} \]
\[ \lor \ LL2_{\text{Restart}} \]
\[ \lor \ LL2_{\text{ReadDisk}} \]
\[ \lor \ LL2_{\text{WriteDisk}} \]
\[ \lor \ LL2_{\text{CorruptRAM}} \]
\[ \lor \ LL2_{\text{CorruptSPCR}} \]

\[ LL2_{\text{Spec}} \triangleq LL2_{\text{Init}} \land [LL2_{\text{Next}}]_{LL2_{\text{Vars}}} \]
3. REFINEMENTS AND INVARIANTS

This section presents two classes of TLA+ modules. First, it contains modules pertaining to the refinement of one spec to another. There are two such modules, one that refines the Memoir-Basic low-level spec to the high-level spec and one that refines the Memoir-Opt low-level spec to the Memoir-Basic low-level spec.

Second, this section includes modules describing invariants maintained by the Memoir-Basic spec. This include both invariants needed for proving the Memoir-Basic refinement and also invariants needed for proving those invariants. No invariants are necessary for proving Memoir-Opt refinement.
3.1 Refinement 1: Mapping Memoir-Basic State to High-Level State

This module describes how to interpret the state of the Memoir-Basic spec as a state of the high-level spec.

This module includes the following definitions:

\[ LL\text{1}HistoryStateBindingAuthenticated \]
\[ LL\text{1}NVRAMHistorySummaryUncorrupted \]
\[ LL\text{1}Refinement \]

EXTENDS MemoirLL1Specification

The \( LL\text{1}HistoryStateBindingAuthenticated \) predicate asserts that a history state binding is authenticated, meaning that the set of observed authenticators includes an authenticator that is a valid MAC of this history state binding.

\[ LL\text{1}HistoryStateBindingAuthenticated(historyStateBinding) \triangleq \exists authenticator \in LL\text{1}ObservedAuthenticators : ValidateMAC(LL\text{1}NVRAM.symmetricKey, historyStateBinding, authenticator) \]

The \( LL\text{1}NVRAMHistorySummaryUncorrupted \) predicate asserts that there exists some state hash that is bound to the history summary in the NVRAM by an authenticated history state binding. This predicate is initially true, and it remains true until a \( LL\text{1}RestrictedCorruption \) action occurs, which makes the predicate false, and it remains false thereafter.

\[ LL\text{1}NVRAMHistorySummaryUncorrupted \triangleq \exists stateHash \in HashType : \]
\[ \text{LET} \]
\[ historyStateBinding \triangleq Hash(LL\text{1}NVRAM.historySummary, stateHash) \]
\[ \text{IN} \]
\[ LL\text{1}HistoryStateBindingAuthenticated(historyStateBinding) \]

The \( LL\text{1}Refinement \) describes the relationship between the Memoir-Basic spec and the high-level spec.

\[ LL\text{1}Refinement \triangleq \]

The high-level available inputs correspond exactly to the Memoir-Basic available inputs, since both are abstractions that model the availability of particular input values to the user.

\[ \land HLAvailableInputs = LL\text{1}AvailableInputs \]

The high-level observed outputs correspond exactly to the Memoir-Basic observed outputs, since both are abstractions that model the set of outputs that the user has so far observed from the operation of the service.

\[ \land HLObservedOutputs = LL\text{1}ObservedOutputs \]

The high-level public and private state is defined in terms of the history summary in the NVRAM. There are two possibilities.

\[ \land \text{IF } LL\text{1}NVRAMHistorySummaryUncorrupted \text{ THEN} \]

First, if the set of observed authenticators contains an authenticator that binds any state hash to the history summary currently in the NVRAM, then the public and private state is any state of the legal type whose hash is so bound. In this case, the service is alive.

\[ \text{LET} \]
\[ refPrivateStateEnc \triangleq SymmetricEncrypt(LL\text{1}NVRAM.symmetricKey, HLPrivateState) \]
\[ refStateHash \triangleq Hash(HLPublicState, refPrivateStateEnc) \]
\[ refHistoryStateBinding \triangleq Hash(LL\text{1}NVRAM.historySummary, refStateHash) \]
\[ \text{IN} \]
\[ \land HLPublicState \in PublicStateType \]
\[ \land HLPrivateState \in PrivateStateType \]
\[ \land LL\text{1}HistoryStateBindingAuthenticated(refHistoryStateBinding) \]
\( \land \text{HL Alive} = \text{TRUE} \)

\text{ELSE}

Second, if the set of observed authenticators does not contain an authenticator that binds any state hash to the history summary currently in the \text{NVRAM}, then the values of the public and private state are equal to their dead states, and the service is not alive.

\( \land \text{HL Public State} = \text{Dead Public State} \)

\( \land \text{HL Private State} = \text{Dead Private State} \)

\( \land \text{HL Alive} = \text{FALSE} \)
3.2 Refinement 2: Mapping Memoir-Opt State to Memoir-Basic State

This module describes how to interpret the state of Memoir-Opt as a state of Memoir-Basic.

This module includes the following definitions:
- LL2HistorySummaryIsSuccessor
- HistorySummariesMatch
- AuthenticatorsMatch
- AuthenticatorSetsMatch
- LL2NVRAMLogicalHistorySummary
- LL2Refinement

EXTENDS MemoirLL2Specification, Sequences

The LL2HistorySummaryIsSuccessor predicate defines, in the Memoir-Opt spec, the conditions under which one history summary is a successor of another history summary with a particular intervening input.

\[ LL2HistorySummaryIsSuccessor(historySummary, previousHistorySummary, input, hashBarrier) \]

\[ \triangleq \]

\[ \text{LET} \]

\[ \text{successorHistorySummary} \triangleq \text{Successor(previousHistorySummary, input, hashBarrier)} \]

\[ \text{checkpointedSuccessorHistorySummary} \triangleq \text{Checkpoint(successorHistorySummary)} \]

\[ \text{IN} \]

\[ \lor historySummary = successorHistorySummary \]

\[ \lor historySummary = checkpointedSuccessorHistorySummary \]

The HistorySummariesMatch predicate defines the conditions under which a history summary in the Memoir-Basic spec semantically matches a history summary in the Memoir-Opt spec.

This requires a recursive definition, but the current version of the prover cannot handle recursive operators, nor can it tractably support proofs using recursive function definitions. Therefore, we define the operator indirectly, by using an assumption. Although Lamport has stated that this approach is “not a satisfactory alternative to recursive definitions,” he has also called it “a reasonable hack to get a proof done.”

CONSTANT HistorySummariesMatch(_ _, _ _)

\[ \text{HistorySummariesMatchRecursion(ll1HistorySummary, ll2HistorySummary, hashBarrier)} \]

\[ \triangleq \]

\[ \exists \text{ input } \in \text{InputType}, \]

\[ \text{previousLL1HistorySummary} \in \text{HashType}, \]

\[ \text{previousLL2HistorySummary} \in \text{HistorySummaryType} : \]

\[ \land \text{HistorySummariesMatch(previousLL1HistorySummary, previousLL2HistorySummary, hashBarrier)} \]

\[ \land ll1HistorySummary = \text{Hash(previousLL1HistorySummary, input)} \]

\[ \land LL2HistorySummaryIsSuccessor(ll2HistorySummary, previousLL2HistorySummary, input, hashBarrier) \]

ASSUME HistorySummariesMatchDefinition \( \triangleq \)

\[ \lor ll1HistorySummary \in \text{HashType}, \]

\[ ll2HistorySummary \in \text{HistorySummaryType}, \]

\[ \text{hashBarrier} \in \text{HashType} : \]

\[ \text{LET} \]

\[ ll2InitialHistorySummary \triangleq \text{[anchor }\rightarrow \text{BaseHashValue, extension }\rightarrow \text{BaseHashValue]} \]

\[ \text{IN} \]

\[ \text{IF } ll2HistorySummary = ll2InitialHistorySummary \]

\[ \text{THEN} \]

\[ \text{HistorySummariesMatch(ll1HistorySummary, ll2HistorySummary, hashBarrier)} = \]
\((\text{ll1HistorySummary} = \text{BaseHashValue})\)

ELSE

\(\text{HistorySummariesMatch}((\text{ll1HistorySummary}, \text{ll2HistorySummary}, \text{hashBarrier}) = \text{HistorySummariesMatchRecursion}((\text{ll1HistorySummary}, \text{ll2HistorySummary}, \text{hashBarrier}))\)

The AuthenticatorsMatch predicate defines the conditions under which an authenticator in the Memoir-Basic spec semantically matches an authenticator in the Memoir-Opt spec.

\[
\text{AuthenticatorsMatch}((\text{ll1Authenticator}, \text{ll2Authenticator}, \text{symmetricKey}, \text{hashBarrier}) \triangleq \exists \text{stateHash} \in \text{HashType}, \\
\text{ll1HistorySummary} \in \text{HashType}, \\
\text{ll2HistorySummary} \in \text{HistorySummaryType} : \\
\text{LET} \\
\text{ll1HistoryStateBinding} \triangleq \text{Hash}(\text{ll1HistorySummary}, \text{stateHash}) \\
\text{ll2HistorySummaryHash} \triangleq \text{Hash}(\text{ll2HistorySummary.} \text{anchor}, \text{ll2HistorySummary.} \text{extension}) \\
\text{ll2HistoryStateBinding} \triangleq \text{Hash}(\text{ll2HistorySummaryHash}, \text{stateHash}) \\
\text{IN} \\
\text{The Memoir-Opt authenticator is a valid MAC of an Memoir-Opt history state binding that binds some Memoir-Opt history summary to some state hash.} \\
\land \text{ValidateMAC(}\text{symmetricKey}, \text{ll2HistoryStateBinding, ll2Authenticator}) \\
\text{The Memoir-Basic authenticator is generated as a MAC of a Memoir-Basic history state binding that binds some Memoir-Basic history summary to the same state hash as the previous conjunct.} \\
\land \text{ll1Authenticator} = \text{GenerateMAC(}\text{symmetricKey}, \text{ll1HistoryStateBinding}) \\
\text{The Memoir-Basic history state binding matches the Memoir-Opt history state binding.} \\
\land \text{HistorySummariesMatch}((\text{ll1HistorySummary}, \text{ll2HistorySummary}, \text{hashBarrier})
\]

The AuthenticatorSetsMatch predicate defines the conditions under which a set of authenticators in the Memoir-Basic spec semantically matches a set of authenticators in the Memoir-Opt spec.

\[
\text{AuthenticatorSetsMatch}((\text{ll1Authenticators}, \text{ll2Authenticators}, \text{symmetricKey}, \text{hashBarrier}) \triangleq \\
\forall \text{ll1Authenticator} \in \text{ll1Authenticators} : \\
\exists \text{ll2Authenticator} \in \text{ll2Authenticators} : \\
\text{AuthenticatorsMatch(} \\
\text{ll1Authenticator, ll2Authenticator, symmetricKey, hashBarrier}) \\
\text{For every authenticator in the Memoir-Opt set, there is some authenticator in the Memoir-Basic set that matches it.} \\
\land \\
\forall \text{ll2Authenticator} \in \text{ll2Authenticators} : \\
\exists \text{ll1Authenticator} \in \text{ll1Authenticators} : \\
\text{AuthenticatorsMatch(} \\
\text{ll1Authenticator, ll2Authenticator, symmetricKey, hashBarrier}) \\
\text{For every authenticator in the Memoir-Basic set, there is some authenticator in the Memoir-Opt set that matches it.}
\]

The LL2NVRAMLLogicalHistorySummary is the history summary that is represented by the state of the NVRAM and the SPCR in the Memoir-Opt spec. The anchor always comes directly from the NVRAM, but the extension only comes from the SPCR if the NVRAM indicates that an
extension is in progress; otherwise, the extension is set to the base hash value. The reason for this is that a
LL2TakeCheckpoint action clears the extensionInProgress flag but is unable to reset the SPCR, so during the time
between a LL2TakeCheckpoint action and a LL2Restart action, the extension is really the base hash value, even though
the SPCR has not yet been reset.

The other interesting aspect of LL2NVRAMLogicalHistorySummary is that if there is an extension in progress but the
SPCR equals the base hash value, then the logical value of the extension is set to a crazy hash value. This is necessary
because if a Restart occurs when the LL2NVRAM.historySummaryAnchor equals the base hash value, the SPCR will
be set to the BaseHashValue, but we don’t want the logical history summary to appear the same as the initial history
summary, which it would if both the anchor and the extension were to equal the base hash value.

CONSTANT CrazyHashValue

ASSUME CrazyHashValueTypeSafe ⊑ CrazyHashValue ∈ HashType

ASSUME CrazyHashValueUnique ⊑
∧ ∀ hashInput1, hashInput2 ∈ HashDomain :
   Hash(hashInput1, hashInput2) ≠ CrazyHashValue
∧ BaseHashValue ≠ CrazyHashValue

LL2NVRAMLogicalHistorySummary ⊑
IF LL2NVRAM.extensionInProgress
THEN
   IF LL2SPCR = BaseHashValue
      THEN
         [anchor ↦ LL2NVRAM.historySummaryAnchor,
          extension ↦ CrazyHashValue]
      ELSE
         [anchor ↦ LL2NVRAM.historySummaryAnchor,
          extension ↦ LL2SPCR]
   ELSE
      [anchor ↦ LL2NVRAM.historySummaryAnchor,
       extension ↦ BaseHashValue]

The LL2Refinement describes the relationship between the 2nd low-level spec (Memoir-Opt) and the 1st low-level spec
(Memoir-Basic).

The following variables are directly equal between the two specs:
   LLxAvailableInputs
   LLxObservedOutputs
   LLxDisk.publicState
   LLxDisk.privateStateEnc
   LLxRAM.publicState
   LLxRAM.privateStateEnc
   LLxNVRAM.symmetricKey

The following variables directly match according to the operators defined above:
   LLxObservedAuthenticators
   LLxDisk.historySummary
   LLxDisk.authenticator
   LLxRAM.historySummary
   LLxRAM.authenticator

Note that the authenticators in the sets of observed authenticators match using the symmetric key in the LL2NVRAM.
By contrast, the authenticators in the disk and RAM match using some unspecified symmetric key, because the values in
the disk and RAM may be set arbitrarily by the user.
The following variable has a more involved matching process, involving both the $LL2NVRAMLogicalHistorySummary$ operator and a match operator:

$LL1NVRAM.historySummary$

For each of the variables that are refined via match predicates rather than through an equality relation, the refinement asserts that the variable has the appropriate type.

For each of the record types ($LL1Disk$, $LL1RAM$, and $LL1NVRAM$), the refinement defines the mapping for each individual field within the record.

\[
LL2Refinement \triangleq \\
\land\ LL1AvailableInputs = LL2AvailableInputs \\
\land\ LL1ObservedOutputs = LL2ObservedOutputs \\
\land\ LL1ObservedAuthenticators \subseteq MACType \\
\land\ AuthenticatorSetsMatch( \\
\quad LL1ObservedAuthenticators, \\
\quad LL2ObservedAuthenticators, \\
\quad LL2NVRAM.symmetricKey, \\
\quad LL2NVRAM.hashBarrier) \\
\land\ LL1Disk \in LL1UntrustedStorageType \\
\land\ LL1Disk.publicState = LL2Disk.publicState \\
\land\ LL1Disk.privateStateEnc = LL2Disk.privateStateEnc \\
\land\ HistorySummariesMatch( \\
\quad LL1Disk.historySummary, \\
\quad LL2Disk.historySummary, \\
\quad LL2NVRAM.hashBarrier) \\
\land\ \exists \text{ symmetricKey} \in SymmetricKeyType : \\
\quad AuthenticatorsMatch( \\
\quad LL1Disk.authenticator, \\
\quad LL2Disk.authenticator, \\
\quad \text{symmetricKey}, \\
\quad LL2NVRAM.hashBarrier) \\
\land\ LL1RAM \in LL1UntrustedStorageType \\
\land\ LL1RAM.publicState = LL2RAM.publicState \\
\land\ LL1RAM.privateStateEnc = LL2RAM.privateStateEnc \\
\land\ HistorySummariesMatch( \\
\quad LL1RAM.historySummary, \\
\quad LL2RAM.historySummary, \\
\quad LL2NVRAM.hashBarrier) \\
\land\ \exists \text{ symmetricKey} \in SymmetricKeyType : \\
\quad AuthenticatorsMatch( \\
\quad LL1RAM.authenticator, \\
\quad LL2RAM.authenticator, \\
\quad \text{symmetricKey}, \\
\quad LL2NVRAM.hashBarrier) \\
\land\ LL1NVRAM \in LL1TrustedStorageType \\
\land\ HistorySummariesMatch( \\
\quad LL1NVRAM.historySummary, \\
\quad LL2NVRAMLogicalHistorySummary, \\
\quad LL2NVRAM.hashBarrier) \\
\land\ LL1NVRAM.symmetricKey = LL2NVRAM.symmetricKey
\]
3.3 Invariants Needed to Prove Memoir-Basic Implementation

This module defines the three correctness invariants needed to prove that the Memoir-Basic spec implements the high-level spec.

EXTENDS MemoirLL1TypeSafety

The UnforgeabilityInvariant states that, for any authenticator residing in the user’s RAM, if the authenticator validates using the symmetric key in the NVRAM, the authenticator is in the set of authenticators that the user observed Memoir to produce.

This is a somewhat boring invariant. It really just extends the assumption in the LL1CorruptRAM action, which constraints the set of authenticators the user can create. If we had written the low-level spec differently, such that this constraint had been expressed in the LL1PerformOperation and LL1RepeatOperation actions instead of the LL1CorruptRAM action, this invariant might not be necessary.

UnforgeabilityInvariant \(\Delta=\)
\[\forall \text{historyStateBinding} \in \text{HashType} : \]
\[\text{ValidateMAC}(\text{LL1NVRAM}.\text{symmetricKey}, \text{historyStateBinding}, \text{LL1RAM}.\text{authenticator}) \Rightarrow \text{LL1RAM}.\text{authenticator} \in \text{LL1ObservedAuthenticators}\]

The InclusionInvariant states that, for any history summary and input that together hash to the history summary in the NVRAM, if this history summary is bound to some public and private state by an authenticated history state binding, then the result of invoking the service with this public state, private state, and input will yield an output that is already in the set of observed outputs and a new history state binding that is already authenticated.

This invariant is needed to show that the LL1RepeatOperation action does not have any ill effects. In particular, this invariant tells us that the output that LL1RepeatOperation will produce is already in LL1ObservedOutputs, and the new authenticator that LL1RepeatOperation will produce makes an assertion that is already being asserted by some authenticator in LL1ObservedAuthenticators.

InclusionInvariant \(\Delta=\)
\[\forall \text{input} \in \text{InputType}, \]
\[\text{historySummary} \in \text{HashType}, \]
\[\text{publicState} \in \text{PublicStateType}, \]
\[\text{privateStateEnc} \in \text{PrivateStateEncType} : \]
\[\text{LET} \]
\[\text{stateHash} \triangleq \text{Hash}(\text{publicState}, \text{privateStateEnc}) \]
\[\text{historyStateBinding} \triangleq \text{Hash}(\text{historySummary}, \text{stateHash}) \]
\[\text{privateState} \triangleq \text{SymmetricDecrypt}(\text{LL1NVRAM}.\text{symmetricKey}, \text{privateStateEnc}) \]
\[\text{sResult} \triangleq \text{Service}(\text{publicState}, \text{privateState}, \text{input}) \]
\[\text{newPrivateStateEnc} \triangleq \]
\[\text{SymmetricEncrypt}(\text{LL1NVRAM}.\text{symmetricKey}, \text{sResult}.\text{newPrivateState}) \]
\[\text{newStateHash} \triangleq \text{Hash}(\text{sResult}.\text{newPublicState}, \text{newPrivateStateEnc}) \]
\[\text{newHistoryStateBinding} \triangleq \text{Hash}(\text{LL1NVRAM}.\text{historySummary}, \text{newStateHash}) \]
\[\text{IN} \]
\[\left(\land \text{LL1NVRAM}.\text{historySummary} = \text{Hash}(\text{historySummary}, \text{input}) \right) \]
\[\land \text{LL1HistoryStateBindingAuthenticated}(\text{historyStateBinding}) \]
\[\Rightarrow \left(\land \text{sResult}.\text{output} \in \text{LL1ObservedOutputs} \right) \]
\[\land \text{LL1HistoryStateBindingAuthenticated}(\text{newHistoryStateBinding}) \]
The UniquenessInvariant states that the history summary in the NVRAM is bound to only one public and private state by an authenticator in the set of observed authenticators. This invariant is used in several places in the implementation proof.

In the NonAdvancementLemma, the property of uniqueness is needed to show that the refined public and private state does not change when the NVRAM and set of observed authenticators does not change. If there were more than one state bound to the history summary in the NVRAM, a low-level stuttering step could lead to a high-level change in state.

In the base case of the Memoir-Basic implementation proof, the uniqueness property is needed to show that the refined state hash corresponds uniquely to the initial state hash.

Within the induction of the Memoir-Basic implementation proof, in the case of LL1PerformOperation, the uniqueness property is used in two places. First, it is needed to show that the public and private state in the arguments to the service correspond to the refined high-level state. Second, it is needed to show that the public and private state produced as the result from the service correspond to the refined high-level primed state.

UniquenessInvariant $\triangleq$
\[
\forall \text{stateHash}_1, \text{stateHash}_2 \in \text{HashType} : \\
\text{LET} \\
\quad \text{historyStateBinding}_1 \triangleq \text{Hash}(\text{LL1NVRAM.historySummary}, \text{stateHash}_1) \\
\quad \text{historyStateBinding}_2 \triangleq \text{Hash}(\text{LL1NVRAM.historySummary}, \text{stateHash}_2) \\
\text{IN} \\
\quad (\wedge \text{LL1HistoryStateBindingAuthenticated(historyStateBinding}_1) \\
\quad \wedge \text{LL1HistoryStateBindingAuthenticated(historyStateBinding}_2)) \\
\Rightarrow \\
\quad \text{stateHash}_1 = \text{stateHash}_2
\]

Collectively, we refer to these three invariants as the correctness invariants for the Memoir-Basic implementation.

CorrectnessInvariants $\triangleq$
\[
\wedge \text{UnforgeabilityInvariant} \\
\wedge \text{InclusionInvariant} \\
\wedge \text{UniquenessInvariant}
\]
3.4 Invariants Needed to Prove Memoir-Basic Invariance

This module defines two supplemental invariants. These are not needed directly by the Memoir-Basic implementation proof, but they are needed by the proofs that the correctness invariants hold.

EXTENDS MemoirLL1CorrectnessInvariants, Naturals

The ExtendedUnforgeabilityInvariant states that the unforgeability property of the authenticator in the RAM also applies to the authenticator on the disk. This is needed to show that the UnforgeabilityInvariant holds through a LL1ReadDisk action.

\[
\text{ExtendedUnforgeabilityInvariant} \triangleq \\
\forall \text{historyStateBinding} \in \text{HashType} : \\
\quad \land \text{ValidateMAC(LL1NVRAM.symmetricKey, historyStateBinding, LL1RAM.authenticator)} \Rightarrow \\
\quad \text{LL1RAM.authenticator} \in \text{LL1ObservedAuthenticators} \\
\land \text{ValidateMAC(LL1NVRAM.symmetricKey, historyStateBinding, LL1Disk.authenticator)} \Rightarrow \\
\land \text{LL1Disk.authenticator} \in \text{LL1ObservedAuthenticators}
\]

The CardinalityInvariant is a little funky. We define a new operator called HashCardinality that indicates the count of hashes needed to produce the supplied hash value. For example, if you create a hash chain, starting with the base hash value, and chain in \( N \) inputs, the cardinality of the resulting hash will be \( N \). Although our low-level spec uses the hash operator only for linear hash chains, the hash cardinality is also defined for arbitrary trees of hashes.

The CardinalityInvariant states that the hash cardinality of the history summary in any observed authenticator is less than or equal to the hash cardinality of the history summary in the NVRAM.

This property is needed to prove that the uniqueness property is an inductive invariant.

CONSTANT HashCardinality(\_)

ASSUME HashCardinalityTypeSafe \triangleq \\
\forall \text{hash} \in \text{HashDomain} : \text{HashCardinality(hash)} \in \text{Nat}

ASSUME BaseHashCardinalityZero \triangleq \text{HashCardinality(BaseHashValue)} = 0

ASSUME InputCardinalityZero \triangleq \\
\forall \text{input} \in \text{InputType} : \text{HashCardinality(input)} = 0

ASSUME HashCardinalityAccumulative \triangleq \\
\forall \text{hash1}, \text{hash2} \in \text{HashDomain} : \\
\text{HashCardinality(Hash(hash1, hash2))} = \\
\text{HashCardinality(hash1) + HashCardinality(hash2) + 1}

CardinalityInvariant \triangleq \\
\forall \text{historySummary} \in \text{HashType}, \text{stateHash} \in \text{HashType} : \\
\text{LET} \\
\quad \text{historyStateBinding} \triangleq \text{Hash(historySummary, stateHash)} \\
\text{IN} \\
\quad (\land \text{LL1NVRAMHistorySummaryUncorrupted} \\
\quad \land \text{LL1HistoryStateBindingAuthenticated(historyStateBinding)}) \\
\Rightarrow \\
\quad \text{HashCardinality(historySummary) \leq HashCardinality(LL1NVRAM.historySummary)}
4. PROOFS
This section presents TLA+ modules that contain proofs. The proofs include type safety of the three specs; lemmas relating to types, invariants, refinement, or implementation; proofs of invariance; and proofs of implementation.
4.1 Proof of Type Safety of the High-Level Spec

This is a very simple proof that shows the high-level spec to be type-safe.

EXTENDS MemoirHLSpecification

THEOREM HLTypeSafe \triangleq \text{HLspec} \Rightarrow \Box \text{HLTypeInvariant}

The top level of the proof is boilerplate TLA+ for an Inv1-style proof. First, we prove that the initial state satisfies HLTypeInvariant. Second, we prove that the HLNext predicate inductively preserves HLTypeInvariant. Third, we use temporal induction to prove that these two conditions satisfy type safety over all behaviors.

\langle 1 \rangle. HLInit \Rightarrow HLTypeInvariant

The base case follows trivially from the definition of HLInit and the assumption that the developer-supplied constants are type-safe.

\langle 2 \rangle. \begin{align*}
&\text{HAVE HLInit} \\
&\text{BY (2), (2) QED}
\end{align*}

\langle 2 \rangle. HLTypeInvariant \land [HLNext]HLVars \Rightarrow HLTypeInvariant'

The induction step is also fairly trivial. We assume the antecedents of the implication, then show that the consequent holds for both HLNext actions.

\langle 2 \rangle. \begin{align*}
&\text{HAVE HLTypeInvariant} \land [HLNext]HLVars \\
&\text{CASE UNCHANGED HLVars}
\end{align*}

Type safety is inductively trivial for a stuttering step.

\langle 2 \rangle. \begin{align*}
&\text{CASE HLMakeInputAvailable} \\
&\text{BY (2), (2) QED}
\end{align*}

Type safety is also trivial for a HLMakeInputAvailable action.

\langle 2 \rangle. \begin{align*}
&\text{CASE HLAdvanceService} \\
&\text{FOR A HLAdvanceService ACTION, WE JUST WALK THROUGH THE DEFINITIONS. TYPE SAFETY FOLLOWS DIRECTLY.}
\end{align*}

\langle 3 \rangle. \begin{align*}
&\text{PICK input} \in \text{HLAvailableInputs} : \text{HLAdvanceService!}(input)!!1 \\
&\text{BY (3), (3) QED}
\end{align*}

\langle 3 \rangle. \begin{align*}
&\text{CASE HLAdvanceService} \\
&\text{BY (3), (3) QED}
\end{align*}

Type safety is also trivial for a HLDie action.

\langle 3 \rangle. \begin{align*}
&\text{CASE HLDie} \\
&\text{BY (3), (3) QED}
\end{align*}

Using the Inv1 proof rule, the base case and the induction step together imply that the invariant always holds.

\langle 2 \rangle. \begin{align*}
&\text{HLTypeInvariant} \land \Box [HLNext]HLVars \Rightarrow \Box HLTypeInvariant
\end{align*}
BY (1)2, Inv1
(2)2, QED
BY (2)1, (1)1 DEF HLSpec
This module states and proves several lemmas that are useful for proving type safety. Since type safety is an important part of the implementation proof, these lemmas also will be used in theorems other than the Memoir-Basic type-safety theorem.

The lemmas in this module are:

- **LL1SubtypeImplicationLemma**
- **LL1InitDefsTypeSafeLemma**
- **LL1PerformOperationDefsTypeSafeLemma**
- **LL1RepeatOperationDefsTypeSafeLemma**
- **InclusionInvariantDefsTypeSafeLemma**
- **CardinalityInvariantDefsTypeSafeLemma**
- **UniquenessInvariantDefsTypeSafeLemma**
- **LL1NVRAMHistorySummaryUncorruptedDefsTypeSafeLemma**
- **LL1RefinementDefsTypeSafeLemma**
- **LL1RefinementPrimeDefsTypeSafeLemma**

**EXTENDS** MemoirLL1Refinement

**LL1SubtypeImplicationLemma** proves that when the **LL1TypeInvariant** holds, the subtypes of **LL1Disk**, **LL1RAM**, and **LL1NVRAM** also hold. This is asserted and proven for both the unprimed and primed states.

The proof itself is completely trivial. It follows directly from the type definitions **LL1UntrustedStorageType** and **LL1TrustedStorageType**.

\[
\text{LL1SubtypeImplication} \triangleq \\
\text{LL1TypeInvariant} \Rightarrow \\
\land \\
\text{LL1Disk}.\text{publicState} \in \text{PublicStateType} \\
\land \\
\text{LL1Disk}.\text{privateStateEnc} \in \text{PrivateStateEncType} \\
\land \\
\text{LL1Disk}.\text{historySummary} \in \text{HashType} \\
\land \\
\text{LL1Disk}.\text{authenticator} \in \text{MACType} \\
\land \\
\text{LL1RAM}.\text{publicState} \in \text{PublicStateType} \\
\land \\
\text{LL1RAM}.\text{privateStateEnc} \in \text{PrivateStateEncType} \\
\land \\
\text{LL1RAM}.\text{historySummary} \in \text{HashType} \\
\land \\
\text{LL1RAM}.\text{authenticator} \in \text{MACType} \\
\land \\
\text{LL1NVRAM}.\text{historySummary} \in \text{HashType} \\
\land \\
\text{LL1NVRAM}.\text{symmetricKey} \in \text{SymmetricKeyType}
\]

**THEOREM** \(\text{LL1SubtypeImplicationLemma} \triangleq\)

\[
\land \\
\text{LL1SubtypeImplication} \\
\land \\
\text{LL1SubtypeImplication}' \\
\langle 1 \rangle \\
\text{ASSUME LL1TypeInvariant} \\
\langle 2 \rangle \\
\text{SUFFICES} \\
\text{PROVE} \\
\land \\
\text{LL1Disk}.\text{publicState} \in \text{PublicStateType} \\
\land \\
\text{LL1Disk}.\text{privateStateEnc} \in \text{PrivateStateEncType} \\
\land \\
\text{LL1Disk}.\text{historySummary} \in \text{HashType} \\
\land \\
\text{LL1Disk}.\text{authenticator} \in \text{MACType} \\
\land \\
\text{LL1RAM}.\text{publicState} \in \text{PublicStateType} \\
\land \\
\text{LL1RAM}.\text{privateStateEnc} \in \text{PrivateStateEncType} \\
\land \\
\text{LL1RAM}.\text{historySummary} \in \text{HashType} \\
\land \\
\text{LL1RAM}.\text{authenticator} \in \text{MACType} \\
\land \\
\text{LL1NVRAM}.\text{historySummary} \in \text{HashType}
\]
\[\land LL1NVRAM \text{.symmetricKey} \in \text{SymmetricKeyType} \]

\text{BY \text{DEF LL1SubtypeImplication}}

(2.2) \text{LL1Disk} \in LL1\text{UntrustedStorageType}
\text{BY \text{(2.1)} \text{DEF LL1TypeInvariant}}

(2.3) \text{LL1RAM} \in LL1\text{UntrustedStorageType}
\text{BY \text{(2.1)} \text{DEF LL1TypeInvariant}}

(2.4) \text{LL1NVRAM} \in LL1\text{TrustedStorageType}
\text{BY \text{(2.1)} \text{DEF LL1TypeInvariant}}

(2.5) \text{LL1Disk} \text{.publicState} \in \text{PublicStateType}
\text{BY \text{(2.2)} \text{DEF LL1UntrustedStorageType}}

(2.6) \text{LL1Disk} \text{.privateStateEnc} \in \text{PrivateStateEncType}
\text{BY \text{(2.2)} \text{DEF LL1UntrustedStorageType}}

(2.7) \text{LL1Disk} \text{.historySummary} \in \text{HashType}
\text{BY \text{(2.2)} \text{DEF LL1UntrustedStorageType}}

(2.8) \text{LL1Disk} \text{. authenticator} \in \text{MACType}
\text{BY \text{(2.2)} \text{DEF LL1UntrustedStorageType}}

(2.9) \text{LL1RAM} \text{.publicState} \in \text{PublicStateType}
\text{BY \text{(2.3)} \text{DEF LL1UntrustedStorageType}}

(2.10) \text{LL1RAM} \text{.privateStateEnc} \in \text{PrivateStateEncType}
\text{BY \text{(2.3)} \text{DEF LL1UntrustedStorageType}}

(2.11) \text{LL1RAM} \text{.historySummary} \in \text{HashType}
\text{BY \text{(2.3)} \text{DEF LL1UntrustedStorageType}}

(2.12) \text{LL1RAM} \text{. authenticator} \in \text{MACType}
\text{BY \text{(2.3)} \text{DEF LL1UntrustedStorageType}}

(2.13) \text{LL1NVRAM} \text{.historySummary} \in \text{HashType}
\text{BY \text{(2.4)} \text{DEF LL1TrustedStorageType}}

(2.14) \text{LL1NVRAM} \text{.symmetricKey} \in \text{SymmetricKeyType}
\text{BY \text{(2.4)} \text{DEF LL1TrustedStorageType}}

(2.15) \text{QED}
\text{BY \text{(2.5, (2.6, (2.7, (2.8, (2.9, (2.10, (2.11, (2.12, (2.13, (2.14)}}

(1.2) \text{LL1SubtypeImplication'}

(2.1) \text{SUFFICES}
\text{ASSUME LL1TypeInvariant'}
\text{PROVE}
\land \text{LL1Disk} \text{.publicState'} \in \text{PublicStateType}
\land \text{LL1Disk} \text{.privateStateEnc'} \in \text{PrivateStateEncType}
\land \text{LL1Disk} \text{.historySummary'} \in \text{HashType}
\land \text{LL1Disk} \text{. authenticator'} \in \text{MACType}
\land \text{LL1RAM} \text{.publicState'} \in \text{PublicStateType}
\land \text{LL1RAM} \text{.privateStateEnc'} \in \text{PrivateStateEncType}
\land \text{LL1RAM} \text{.historySummary'} \in \text{HashType}
\land \text{LL1RAM} \text{. authenticator'} \in \text{MACType}
\land \text{LL1NVRAM} \text{.historySummary'} \in \text{HashType}
\land \text{LL1NVRAM} \text{.symmetricKey'} \in \text{SymmetricKeyType}
\text{BY \text{DEF LL1SubtypeImplication}}

(2.2) \text{LL1Disk'} \in \text{LL1UntrustedStorageType}
\text{BY \text{(2.1)} \text{DEF LL1TypeInvariant}}

(2.3) \text{LL1RAM'} \in \text{LL1UntrustedStorageType}
\text{BY \text{(2.1)} \text{DEF LL1TypeInvariant}}

(2.4) \text{LL1NVRAM'} \in \text{LL1TrustedStorageType}
\text{BY \text{(2.1)} \text{DEF LL1TypeInvariant}}

(2.5) \text{LL1Disk} \text{.publicState'} \in \text{PublicStateType}
THEOREM LL1InitDefsTypeSafeLemma ≡
∀ symmetricKey ∈ SymmetricKeyType :
LET

InitialPrivateStateEnc ≡ SymmetricEncrypt(symmetricKey, InitialPrivateState)
InitialStateHash ≡ Hash(InitialPublicState, initialPrivateStateEnc)
InitialHistoryStateBinding ≡ Hash(BaseHashValue, initialStateHash)
InitialAuthenticator ≡ GenerateMAC(symmetricKey, initialHistoryStateBinding)
InitialUntrustedStorage ≡ [PublicState ↦ InitialPublicState,
PrivateStateEnc ↦ initialPrivateStateEnc,
historySummary ↦ BaseHashValue,
authenticator ↦ initialAuthenticator]
InitialTrustedStorage ≡ [
historySummary ↦ BaseHashValue,
symmetricKey ↦ symmetricKey] IN

∧ initialPrivateStateEnc ∈ PrivateStateEncType
∧ initialStateHash ∈ HashType
∧ initialHistoryStateBinding ∈ HashType
∧ initialAuthenticator ∈ MACType
∧ initialUntrustedStorage ∈ LL1UntrustedStorageType
∧ initialTrustedStorage ∈ LL1TrustedStorageType

⟨1⟩. TAKE symmetricKey ∈ SymmetricKeyType
⟨1⟩ initialPrivateStateEnc ≡ SymmetricEncrypt(symmetricKey, InitialPrivateState)
⟨1⟩ initialStateHash ≡ Hash(InitialPublicState, initialPrivateStateEnc)
⟨1⟩ initialHistoryStateBinding ≡ Hash(BaseHashValue, initialStateHash)
\(1\) \(\text{initialAuthenticator} \triangleq \text{GenerateMAC}(\text{symmetricKey}, \text{initialHistoryStateBinding})\)

\(1\) \(\text{initialUntrustedStorage} \triangleq \begin{array}{l}
\text{publicState} \mapsto \text{InitialPublicState}, \\
\text{privateStateEnc} \mapsto \text{initialPrivateStateEnc}, \\
\text{historySummary} \mapsto \text{BaseHashValue}, \\
\text{authenticator} \mapsto \text{initialAuthenticator}
\end{array}\)

\(1\) \(\text{initialTrustedStorage} \triangleq \begin{array}{l}
\text{historySummary} \mapsto \text{BaseHashValue}, \\
\text{symmetricKey} \mapsto \text{symmetricKey}
\end{array}\)

\(1\) \(\text{hide def}\ \text{initialPrivateStateEnc}, \text{initialStateHash}, \text{initialAuthenticator}, \text{initialUntrustedStorage}, \text{initialTrustedStorage}\)

\(1\) \(2\). \(\text{initialPrivateStateEnc} \in \text{PrivateStateEncType}\)

\(2\) \(1\). \(\text{symmetricKey} \in \text{SymmetricKeyType}\)

\(2\) \(2\). \(\text{InitialPrivateState} \in \text{PrivateStateType}\)

\(2\) \(3\). \(\text{QED}\)

\(2\) \(3\). \(\text{QED}\)

\(2\) \(3\). \(\text{QED}\)

\(2\) \(4\). \(\text{initialHistoryStateBinding} \in \text{HashType}\)

\(2\) \(1\). \(\text{BaseHashValue} \in \text{HashDomain}\)

\(2\) \(3\). \(\text{QED}\)

\(2\) \(3\). \(\text{QED}\)

\(2\) \(4\). \(\text{initialHistoryStateBinding} \in \text{HashType}\)

\(2\) \(1\). \(\text{symmetricKey} \in \text{SymmetricKeyType}\)

\(2\) \(2\). \(\text{initialHistoryStateBinding} \in \text{HashType}\)

\(2\) \(3\). \(\text{QED}\)

\(2\) \(3\). \(\text{QED}\)

\(2\) \(6\). \(\text{initialUntrustedStorage} \in \text{LL1UntrustedStorageType}\)

\(2\) \(1\). \(\text{InitialPublicState} \in \text{PublicStateType}\)

\(2\) \(2\). \(\text{initialPrivateStateEnc} \in \text{PrivateStateEncType}\)

\(2\) \(3\). \(\text{QED}\)

\(2\) \(3\). \(\text{QED}\)

\(2\) \(3\). \(\text{QED}\)

\(2\) \(5\). \(\text{initialAuthenticator} \in \text{MACType}\)

\(2\) \(1\). \(\text{symmetricKey} \in \text{SymmetricKeyType}\)

\(2\) \(2\). \(\text{initialHistoryStateBinding} \in \text{HashType}\)

\(2\) \(3\). \(\text{QED}\)

\(2\) \(3\). \(\text{QED}\)

\(2\) \(6\). \(\text{initialUntrustedStorage} \in \text{LL1UntrustedStorageType}\)

\(2\) \(1\). \(\text{InitialPublicState} \in \text{PublicStateType}\)

\(2\) \(2\). \(\text{initialPrivateStateEnc} \in \text{PrivateStateEncType}\)

\(2\) \(3\). \(\text{QED}\)

\(2\) \(3\). \(\text{QED}\)
\[\forall \text{input} \in \text{LL1AvailableInputs} : \]
\[\text{LL1TypeInvariant} \Rightarrow \]
\[\text{LET} \]
\[\text{stateHash} \triangleq \text{Hash}(\text{LL1RAM.publicState}, \text{LL1RAM.privateStateEnc})\]
\[\text{historyStateBinding} \triangleq \text{Hash}(\text{LL1RAM.historySummary}, \text{stateHash})\]
\[\text{privateState} \triangleq \text{SymmetricDecrypt}(\text{LL1NVRAM.symmetricKey}, \text{LL1RAM.privateStateEnc})\]
\[\text{sResult} \triangleq \text{Service}(\text{LL1RAM.publicState}, \text{privateState}, \text{input})\]
\[\text{newPrivateStateEnc} \triangleq \text{SymmetricEncrypt}(\text{LL1NVRAM.symmetricKey}, \text{sResult.newPrivateState})\]
\[\text{newHistorySummary} \triangleq \text{Hash}(\text{LL1NVRAM.historySummary}, \text{input})\]
\[\text{newStateHash} \triangleq \text{Hash}(\text{sResult.newPublicState}, \text{newPrivateStateEnc})\]
\[\text{newHistoryStateBinding} \triangleq \text{Hash}(\text{newHistorySummary}, \text{newStateHash})\]
\[\text{newAuthenticator} \triangleq \text{GenerateMAC}(\text{LL1NVRAM.symmetricKey}, \text{newHistoryStateBinding})\]
\[\text{IN} \]
\[\land \text{stateHash} \in \text{HashType}\]
\[\land \text{historyStateBinding} \in \text{HashType}\]
\[\land \text{privateState} \in \text{PrivateStateType}\]
\[\land \text{sResult} \in \text{ServiceResultType}\]
\[\land \text{sResult.newPublicState} \in \text{PublicStateType}\]
\[\land \text{sResult.newPrivateState} \in \text{PrivateStateType}\]
\[\land \text{sResult.output} \in \text{OutputType}\]
\[\land \text{newPrivateStateEnc} \in \text{PrivateStateEncType}\]
\[\land \text{newHistorySummary} \in \text{HashType}\]
\[\land \text{newStateHash} \in \text{HashType}\]
\[\land \text{newHistoryStateBinding} \in \text{HashType}\]
\[\land \text{newAuthenticator} \in \text{MACType}\]

\[\begin{align*}
(1) & \quad \text{stateHash} \triangleq \text{Hash}(\text{LL1RAM.publicState}, \text{LL1RAM.privateStateEnc}) \\
(1) & \quad \text{historyStateBinding} \triangleq \text{Hash}(\text{LL1RAM.historySummary}, \text{stateHash}) \\
(1) & \quad \text{privateState} \triangleq \text{SymmetricDecrypt}(\text{LL1NVRAM.symmetricKey}, \text{LL1RAM.privateStateEnc}) \\
(1) & \quad \text{sResult} \triangleq \text{Service}(\text{LL1RAM.publicState}, \text{privateState}, \text{input}) \\
(1) & \quad \text{newPrivateStateEnc} \triangleq \\
& \quad \text{SymmetricEncrypt}(\text{LL1NVRAM.symmetricKey}, \text{sResult.newPrivateState})
\end{align*}\]
(1) newHistorySummary \(\triangleq\) Hash(LL1NVRAM.historySummary, input)
(2) newPrivateStateEnc \(\triangleq\) Hash(sResult, newPublicState, newPrivateStateEnc)
(3) newHistoryStateBinding \(\triangleq\) Hash(newHistorySummary, newStateHash)
(4) newAuthenticator \(\triangleq\) GenerateMAC(LL1NVRAM.symmetricKey, newHistoryStateBinding)
(5) HIDE DEF stateHash, historyStateBinding, privateState, sResult, newPrivateStateEnc, 
    newHistorySummary, newStateHash, newHistoryStateBinding, newAuthenticator
(6) 2. HAVE LL1TypeInvariant
(7) 3. stateHash \(\in\) HashType
    (8) 2.1. \(\land\) LL1RAM.publicState \(\in\) PublicStateType 
        \(\land\) LL1RAM.privateStateEnc \(\in\) PrivateStateEncType 
        BY (1)2, LL1SubtypeImplicationLemma DEF LL1SubtypeImplication
    (9) 2.2. \(\land\) LL1RAM.publicState \(\in\) HashDomain 
        \(\land\) LL1RAM.privateStateEnc \(\in\) HashDomain 
        BY (2)1 DEF HashDomain
    (10) 3. QED
        BY (2)2, HashTypeSafe DEF stateHash
(8) 4. historyStateBinding \(\in\) HashType
    (9) 2.1. LL1RAM.historySummary \(\in\) HashDomain
    (10) 3.1. LL1RAM.historySummary \(\in\) HashType 
        BY (1)2, LL1SubtypeImplicationLemma DEF LL1SubtypeImplication
    (11) 3.2. QED 
        BY (3)1 DEF HashDomain
    (12) 2. stateHash \(\in\) HashDomain 
        BY (1)3 DEF HashDomain 
    (13) 3. QED 
        BY (2)1, (12)2, HashTypeSafe DEF historyStateBinding
(14) 5. privateState \(\in\) PrivateStateType
    (15) 2.1. \(\land\) LL1NVRAM.symmetricKey \(\in\) SymmetricKeyType 
        \(\land\) LL1RAM.privateStateEnc \(\in\) PrivateStateEncType 
        BY (1)2, LL1SubtypeImplicationLemma DEF LL1SubtypeImplication
    (16) 2. QED 
        BY (2)1, SymmetricDecryptionTypeSafe DEF privateState
(17) 6. sResult \(\in\) ServiceResultType
    (18) 2.1. LL1RAM.publicState \(\in\) PublicStateType 
        BY (1)2, LL1SubtypeImplicationLemma DEF LL1SubtypeImplication
    (19) 2. privateState \(\in\) PrivateStateType 
        BY (1)5 
    (20) 3. input \(\in\) InputType 
        (21) 3.1. LL1AvailableInputs \(\subseteq\) InputType 
            BY (1)2 DEF LL1TypeInvariant 
        (22) 3.2. QED 
            BY (1)1, (3)1 
    (23) 4. QED 
        BY (2)1, (19)2, (20)3, ServiceTypeSafe DEF sResult
(24) 7. \(\land\) sResult.newPublicState \(\in\) PublicStateType 
    (25) \(\land\) sResult.newPrivateState \(\in\) PrivateStateType 
    (26) \(\land\) sResult.output \(\in\) OutputType 
        BY (1)6 DEF ServiceResultType
(27) 8. newPrivateStateEnc \(\in\) PrivateStateEncType 
    (28) 2.1. LL1NVRAM.symmetricKey \(\in\) SymmetricKeyType 
        BY (1)2, LL1SubtypeImplicationLemma DEF LL1SubtypeImplication 
    (29) 2. sResult.newPrivateState \(\in\) PrivateStateType
LL₁RepeatOperationDefsTypeSafeLemma proves that the definitions within the let of the LL₁RepeatOperation action all have the appropriate type. This is a trivial proof that merely walks through the definitions.

THEOREM LL₁RepeatOperationDefsTypeSafeLemma \[\Delta\]
\[\forall \text{input} \in \text{LL₁AvailableInputs} :\]
$$LL1TypeInvariant \Rightarrow$$

LET

\[\text{stateHash} \triangleq \text{Hash}(\text{LL1RAM.publicState, LL1RAM.privateStateEnc})\]
\[\text{historyStateBinding} \triangleq \text{Hash}(\text{LL1RAM.historySummary, stateHash})\]
\[\text{privateState} \triangleq \text{SymmetricDecrypt}(\text{LL1NVRAM.symmetricKey, LL1RAM.privateStateEnc})\]
\[sResult \triangleq \text{Service}(\text{LL1RAM.publicState, privateState, input})\]
\[\text{newPrivateStateEnc} \triangleq \text{SymmetricEncrypt}(\text{LL1NVRAM.symmetricKey, sResult.newPrivateState})\]
\[\text{newStateHash} \triangleq \text{Hash}(sResult.newPublicState, \text{newPrivateStateEnc})\]
\[\text{newHistoryStateBinding} \triangleq \text{Hash}(\text{LL1NVRAM.historySummary, newStateHash})\]
\[\text{newAuthenticator} \triangleq \text{GenerateMAC}(\text{LL1NVRAM.symmetricKey, newHistoryStateBinding})\]

IN

\[\land \text{stateHash} \in \text{HashType}\]
\[\land \text{historyStateBinding} \in \text{HashType}\]
\[\land \text{privateState} \in \text{PrivateKeyType}\]
\[\land \text{sResult} \in \text{ServiceResultType}\]
\[\land \text{sResult.newPublicState} \in \text{PublicKeyType}\]
\[\land \text{sResult.newPrivateState} \in \text{PrivateKeyType}\]
\[\land \text{sResult.output} \in \text{OutputType}\]
\[\land \text{newPrivateStateEnc} \in \text{PrivateKeyEncType}\]
\[\land \text{newStateHash} \in \text{HashType}\]
\[\land \text{newHistoryStateBinding} \in \text{HashType}\]
\[\land \text{newAuthenticator} \in \text{MACType}\]

\(\langle 1 \rangle1. \text{TAKe} \text{input} \in \text{LL1AvailableInputs}\)
\(\langle 1 \rangle \text{stateHash} \triangleq \text{Hash}(\text{LL1RAM.publicState, LL1RAM.privateStateEnc})\)
\(\langle 1 \rangle \text{historyStateBinding} \triangleq \text{Hash}(\text{LL1RAM.historySummary, stateHash})\)
\(\langle 1 \rangle \text{privateState} \triangleq \text{SymmetricDecrypt}(\text{LL1NVRAM.symmetricKey, LL1RAM.privateStateEnc})\)
\(\langle 1 \rangle \text{sResult} \triangleq \text{Service}(\text{LL1RAM.publicState, privateState, input})\)
\(\langle 1 \rangle \text{newPrivateStateEnc} \triangleq \text{SymmetricEncrypt}(\text{LL1NVRAM.symmetricKey, sResult.newPrivateState})\)
\(\langle 1 \rangle \text{newStateHash} \triangleq \text{Hash}(sResult.newPublicState, \text{newPrivateStateEnc})\)
\(\langle 1 \rangle \text{newHistoryStateBinding} \triangleq \text{Hash}(\text{LL1NVRAM.historySummary, newStateHash})\)
\(\langle 1 \rangle \text{newAuthenticator} \triangleq \text{GenerateMAC}(\text{LL1NVRAM.symmetricKey, newHistoryStateBinding})\)
\(\langle 1 \rangle2. \text{HIDE} \text{def} \text{stateHash, historyStateBinding, privateState, sResult, newPrivateStateEnc, newStateHash, newHistoryStateBinding, newAuthenticator}\)
\(\langle 1 \rangle3. \text{stateHash} \in \text{HashType}\)
\(\langle 2 \rangle1. \land \text{LL1RAM.publicState} \in \text{PublicKeyType}\)
\(\land \text{LL1RAM.privateStateEnc} \in \text{PrivateKeyEncType}\)
\(\text{by} \langle 1 \rangle2, \text{LL1SubtypeImplicationLemma}\text{def} \text{LL1SubtypeImplication}\)
\(\langle 2 \rangle2. \land \text{LL1RAM.publicState} \in \text{HashDomain}\)
\(\land \text{LL1RAM.privateStateEnc} \in \text{HashDomain}\)
\(\text{by} \langle 2 \rangle1 \text{def} \text{HashDomain}\)
\(\langle 2 \rangle3. \text{QED}\)
\(\text{by} \langle 2 \rangle2, \text{HashTypeSafe}\text{def} \text{stateHash}\)
\(\langle 4 \rangle4. \text{historyStateBinding} \in \text{HashType}\)
\(\langle 2 \rangle1. \text{LL1RAM.historySummary} \in \text{HashDomain}\)
\(\langle 3 \rangle1. \text{LL1RAM.historySummary} \in \text{HashType}\)
\(\text{by} \langle 1 \rangle2, \text{LL1SubtypeImplicationLemma}\text{def} \text{LL1SubtypeImplication}\)
\(\langle 3 \rangle2. \text{QED}\)
\(\text{by} \langle 3 \rangle1 \text{def} \text{HashDomain}\)
\(\langle 2 \rangle2. \text{stateHash} \in \text{HashDomain}\)
by (1)3 def HashDomain
(2)3. qed
by (2)1, (2)2, HashTypeSafe
def historyStateBinding
(1)5. privateState ∈ PrivateStateType
(2)1. ∧ LL1NVRAM.symmetricKey ∈ SymmetricKeyType
∧ LL1RAM.privateStateEnc ∈ PrivateStateEncType
by (1)2, LL1SubtypeImplicationLemma def LL1SubtypeImplication
(2)2. qed
by (2)1, SymmetricDecryptionTypeSafe
def privateState
(1)6. sResult ∈ ServiceResultType
(2)1. LL1RAM.publicState ∈ PublicStateType
by (1)2, LL1SubtypeImplicationLemma def LL1SubtypeImplication
(2)2. privateState ∈ PrivateStateType
by (1)5
(2)3. input ∈ InputType
(3)1. LL1AvailableInputs ⊆ InputType
by (1)2 def LL1TypeInvariant
(3)2. qed
by (1)1, (3)1
(2)4. qed
by (2)1, (2)2, (2)3, ServiceTypeSafe
def sResult
(1)7. ∧ sResult.newPublicState ∈ PublicStateType
∧ sResult.newPrivateState ∈ PrivateStateType
∧ sResult.output ∈ OutputType
by (1)6 def ServiceResultType
(1)8. newPrivateStateEnc ∈ PrivateStateEncType
(2)1. LL1NVRAM.symmetricKey ∈ SymmetricKeyType
by (1)2, LL1SubtypeImplicationLemma def LL1SubtypeImplication
(2)2. sResult.newPrivateState ∈ PrivateStateType
by (1)7
(2)3. qed
by (2)1, (2)2, SymmetricEncryptionTypeSafe
def newPrivateStateEnc
(1)9. newStateHash ∈ HashType
(2)1. sResult.newPublicState ∈ HashDomain
(3)1. sResult.newPublicState ∈ PublicStateType
by (1)7
(3)2. qed
by (3)1 def HashDomain
(2)2. newPrivateStateEnc ∈ HashDomain
by (1)8 def HashDomain
(2)3. qed
by (2)1, (2)2, HashTypeSafe
def newStateHash
(1)10. newHistoryStateBinding ∈ HashType
(2)1. LL1NVRAM.historySummary ∈ HashDomain
(3)1. LL1NVRAM.historySummary ∈ HashType
by (1)2, LL1SubtypeImplicationLemma def LL1SubtypeImplication
(3)2. qed
by (3)1 def HashDomain
(2)2. newStateHash ∈ HashDomain
by (1)9 def HashDomain
(2)3. qed
by (2)1, (2)2, HashTypeSafe
def newHistoryStateBinding

62
The InclusionInvariantDefsTypeSafeLemma proves that the definitions within the let of the InclusionInvariant all have the appropriate type. This is a trivial proof that merely walks through the definitions.

**THEOREM** InclusionInvariantDefsTypeSafeLemma

\[
\forall \text{ input} \in \text{InputType}, \text{ historySummary} \in \text{HashType}, \text{ publicState} \in \text{PublicStateType}, \text{ privateStateEnc} \in \text{PrivateStateEncType} : \text{ LL1TypeInvariant} \Rightarrow \\
\text{LE}
\]

\[
\begin{align*}
\text{stateHash} & \overset{\Delta}{=} \text{Hash}(\text{publicState}, \text{privateStateEnc}) \\
\text{historyStateBinding} & \overset{\Delta}{=} \text{Hash}(\text{historySummary}, \text{stateHash}) \\
\text{privateState} & \overset{\Delta}{=} \text{SymmetricDecrypt}\left(\text{LL1NVRAM.symmetricKey}, \text{privateStateEnc}\right) \\
\text{sResult} & \overset{\Delta}{=} \text{Service}(\text{publicState}, \text{privateState}, \text{input}) \\
\text{newPrivateStateEnc} & \overset{\Delta}{=} \\
& \text{SymmetricEncrypt}\left(\text{LL1NVRAM.symmetricKey}, \text{sResult}\text{.newPrivateState}\right) \\
\text{newStateHash} & \overset{\Delta}{=} \text{Hash}\left(\text{sResult}\text{.newPublicState}, \text{newPrivateStateEnc}\right) \\
\text{newHistoryStateBinding} & \overset{\Delta}{=} \text{Hash}\left(\text{LL1NVRAM.historySummary}, \text{newStateHash}\right)
\end{align*}
\]

\[
\in \\
\begin{align*}
& \text{ stateHash} \in \text{HashType} \\
& \text{ historyStateBinding} \in \text{HashType} \\
& \text{ privateState} \in \text{PrivateStateType} \\
& \text{ sResult} \in \text{ServiceResultType} \\
& \text{ sResult}\text{.newPublicState} \in \text{PublicStateType} \\
& \text{ sResult}\text{.newPrivateState} \in \text{PrivateStateType} \\
& \text{ sResult}\text{.output} \in \text{OutputType} \\
& \text{ newPrivateStateEnc} \in \text{PrivateStateEncType} \\
& \text{ newStateHash} \in \text{HashType} \\
& \text{ newHistoryStateBinding} \in \text{HashType}
\end{align*}
\]

\[\langle 1 \rangle. \text{TAK}\]

\[
\begin{align*}
\text{ input} & \in \text{InputType}, \text{ historySummary} \in \text{HashType}, \text{ publicState} \in \text{PublicStateType}, \text{ privateStateEnc} \in \text{PrivateStateEncType} \\
\text{ stateHash} & \overset{\Delta}{=} \text{Hash}(\text{publicState}, \text{privateStateEnc}) \\
\text{ historyStateBinding} & \overset{\Delta}{=} \text{Hash}(\text{historySummary}, \text{stateHash}) \\
\text{ privateState} & \overset{\Delta}{=} \text{SymmetricDecrypt}\left(\text{LL1NVRAM.symmetricKey}, \text{privateStateEnc}\right) \\
\text{ sResult} & \overset{\Delta}{=} \text{Service}(\text{publicState}, \text{privateState}, \text{input}) \\
\text{ newPrivateStateEnc} & \overset{\Delta}{=} \\
& \text{SymmetricEncrypt}\left(\text{LL1NVRAM.symmetricKey}, \text{sResult}\text{.newPrivateState}\right)
\end{align*}
\]

63
\[\text{newStateHash} \triangleq \text{Hash}(\text{sResult}.\text{newPublicState}, \text{newPrivateStateEnc})\]

\[\text{newHistoryStateBinding} \triangleq \text{Hash}(\text{LL1NVRAM}.\text{historySummary}, \text{newStateHash})\]

\{1\) \text{HIDE DEF stateHash, historyStateBinding, privateState, sResult, newPrivateStateEnc, newStateHash, newHistoryStateBinding}\}

\{2\) \text{HAVE LL1TypeInvariant}\}

\{3\) \text{stateHash} \in \text{HashType}\]

\{2\} \land \text{publicState} \in \text{PublicStateType}
\land \text{privateStateEnc} \in \text{PrivateStateEncType}

\text{BY (1)1}\]

\{2\} \land \text{publicState} \in \text{HashDomain}
\land \text{privateStateEnc} \in \text{HashDomain}

\text{BY (2)1 DEF HashDomain}\]

\{2\) \text{QED}\]

\text{BY (2)2, HashTypeSafeDEF stateHash}\]

\{4\) \text{historyStateBinding} \in \text{HashType}\]

\{2\} \text{historySummary} \in \text{HashDomain}

\text{BY (1)1}\]

\{3\) \text{QED}\]

\text{BY (3)1 DEF HashDomain}\]

\{2\) \text{stateHash} \in \text{HashDomain}\]

\text{BY (1)3 DEF HashDomain}\]

\{2\) \text{QED}\]

\text{BY (2)2, (2)2, HashTypeSafeDEF historyStateBinding}\]

\{5\) \text{privateState} \in \text{PrivateStateType}\]

\{2\} \text{LL1NVRAM}.\text{symmetricKey} \in \text{SymmetricKeyType}

\text{BY (1)2, LL1SubtypeImplicationLemmaDEF LL1SubtypeImplication}\]

\{2\) \text{privateStateEnc} \in \text{PrivateStateEncType}\]

\text{BY (1)1}\]

\{2\) \text{QED}\]

\text{BY (2)1, (2)2, SymmetricDecryptionTypeSafeDEF privateState}\]

\{6\) \text{sResult} \in \text{ServiceResultType}\]

\{2\} \text{publicState} \in \text{PublicStateType}

\text{BY (1)1}\]

\{2\) \text{privateState} \in \text{PrivateStateType}

\text{BY (1)5}\]

\{2\) \text{input} \in \text{InputType}\]

\{3\} \text{LL1AvailableInputs} \subseteq \text{InputType}

\text{BY (1)2 DEF LL1TypeInvariant}\]

\{3\} \text{QED}\]

\text{BY (1)1, (3)1}\]

\{2\) \text{QED}\]

\text{BY (2)2, (2)2, ServiceTypeSafeDEF sResult}\]

\{7\) \land \text{sResult}.\text{newPublicState} \in \text{PublicStateType}
\land \text{sResult}.\text{newPrivateState} \in \text{PrivateStateType}
\land \text{sResult}.\text{output} \in \text{OutputType}\]

\text{BY (1)6 DEF ServiceResultType}\]

\{8\) \text{newPrivateStateEnc} \in \text{PrivateStateEncType}\]

\{2\} \text{LL1NVRAM}.\text{symmetricKey} \in \text{SymmetricKeyType}

\text{BY (1)2, LL1SubtypeImplicationLemmaDEF LL1SubtypeImplication}\]

\{2\) \text{sResult}.\text{newPrivateState} \in \text{PrivateStateType}\]

\text{BY (1)7}\]
The `CardinalityInvariantDefsTypeSafeLemma` proves that the definition within the `let` of the `CardinalityInvariant` has the appropriate type. This is a trivial proof that merely walks through the definitions.
The `UniquenessInvariantDefsTypeSafeLemma` proves that the definitions within the `let` of the `UniquenessInvariant` all have the appropriate type. This is a trivial proof that merely walks through the definitions.

**THEOREM UniquenessInvariantDefsTypeSafeLemma** 
\( \forall \text{stateHash}_1, \text{stateHash}_2 \in \text{HashType} : \)
\[ \text{LL1TypeInvariant} \Rightarrow \]
\[ \text{LET} \]
\[ \text{historyStateBinding}_1 \triangleq \text{Hash}(\text{LL1NVRAM.historySummary}, \text{stateHash}_1) \]
\[ \text{historyStateBinding}_2 \triangleq \text{Hash}(\text{LL1NVRAM.historySummary}, \text{stateHash}_2) \]
\[ \text{IN} \]
\[ \land \text{historyStateBinding}_1 \in \text{HashType} \]
\[ \land \text{historyStateBinding}_2 \in \text{HashType} \]
\[ \{1\}. \text{TAKE stateHash}_1, \text{stateHash}_2 \in \text{HashType} \]
\[ \{1\} \text{historyStateBinding}_1 \triangleq \text{Hash}(\text{LL1NVRAM.historySummary}, \text{stateHash}_1) \]
\[ \{1\} \text{historyStateBinding}_2 \triangleq \text{Hash}(\text{LL1NVRAM.historySummary}, \text{stateHash}_2) \]
\[ \{1\} \text{HIDE DEF historyStateBinding}_1, \text{historyStateBinding}_2 \]
\[ \{1\}. \text{HAVE LL1TypeInvariant} \]
\[ \{1\}.3. \text{LL1NVRAM.historySummary} \in \text{HashDomain} \]
\[ \{2\}. \text{LL1NVRAM.historySummary} \in \text{HashType} \]
\[ \{3\}. \text{LL1TypeInvariant} \]
\[ \text{BY (1)2} \]
\[ \{3\}. \text{QED} \]
\[ \text{BY (3)1, LL1SubtypeImplicationLemma} \text{DEF LL1SubtypeImplication} \]
\[ \{2\}. \text{QED} \]
\[ \text{BY (2)1 DEF HashDomain} \]
\[ \{1\}.4. \text{historyStateBinding}_1 \in \text{HashType} \]
\[ \{2\}. \text{stateHash}_1 \in \text{HashDomain} \]
\[ \{3\}. \text{stateHash}_1 \in \text{HashType} \]
\[ \text{BY (1)1} \]
\[ \{3\}. \text{QED} \]
\[ \text{BY (3)1 DEF HashDomain} \]
\[ \{2\}. \text{QED} \]
\[ \text{BY (1),3, (2)1, HashTypeSafe} \text{DEF historyStateBinding}_1 \]
\[ \{1\}.5. \text{historyStateBinding}_2 \in \text{HashType} \]
\[ \{2\}. \text{stateHash}_2 \in \text{HashDomain} \]
\[ \{3\}. \text{stateHash}_2 \in \text{HashType} \]
\[ \text{BY (1)1} \]
\[ \{3\}. \text{QED} \]
\[ \text{BY (3)1 DEF HashDomain} \]
\[ \{2\}. \text{QED} \]
\[ \text{BY (1),3, (2)1, HashTypeSafe} \text{DEF historyStateBinding}_2 \]
\[ \{1\}.6. \text{QED} \]
\[ \text{BY (1)4, (1)5} \]
\[ \text{DEF historyStateBinding}_1, \text{historyStateBinding}_2 \]

The `LL1RefinementDefsTypeSafeLemma` proves that the definitions within the `let` of the `LL1Refinement` definition all have the appropriate type in the unprimed state. This is a trivial proof that merely walks through the definitions.
The **LL1RefinementPrimeDefsTypeSafeLemma** proves that the definitions within the `let` of the `LL1Refinement` definition all have the appropriate type in the primed state. This is a trivial proof that merely walks through the definitions.
LET
  refPrivateStateEnc ≜ SymmetricEncrypt(LL1NVRAM.symmetricKey, HLPrivateState)
  refStateHash ≜ Hash(HLPublicState, refPrivateStateEnc)
  refHistoryStateBinding ≜ Hash(LL1NVRAM.historySummary, refStateHash)
IN
  ∧ refPrivateStateEnc' ∈ PrivateStateEncType
  ∧ refStateHash' ∈ HashType
  ∧ refHistoryStateBinding' ∈ HashType
(1) refPrivateStateEnc ≜ SymmetricEncrypt(LL1NVRAM.symmetricKey, HLPrivateState)
(1) refStateHash ≜ Hash(HLPublicState, refPrivateStateEnc)
(1) refHistoryStateBinding ≜ Hash(LL1NVRAM.historySummary, refStateHash)
(1) HIDE DEF refPrivateStateEnc, refStateHash
(1)1. HAVE LL1Refinement' ∧ LL1TypeInvariant'
(1)2. refPrivateStateEnc' ∈ PrivateStateEncType
  (2)1. LL1NVRAM.symmetricKey' ∈ SymmetricKeyType
    BY (1)1, LL1SubtypeImplicationLemma DEF LL1SubtypeImplication
  (2)2. HLPublicState' ∈ PrivateStateType
    BY (1)1, ConstantsTypeSafe DEF LL1Refinement
  (2)3. QED
    BY (2)1, (2)2, SymmetricEncryptionTypeSafe DEF refPrivateStateEnc
(1)3. refStateHash' ∈ HashType
  (2)1. HLPublicState' ∈ HashDomain
    (3)1. HLPublicState' ∈ PublicStateType
      BY (1)1, ConstantsTypeSafe DEF LL1Refinement
    (3)2. QED
      BY (3)1 DEF HashDomain
    (2)2. refPrivateStateEnc' ∈ HashDomain
      BY (1)2 DEF HashDomain
    (2)3. QED
      BY (2)1, (2)2, HashTypeSafe DEF refStateHash
(1)4. refHistoryStateBinding' ∈ HashType
  (2)1. LL1NVRAM.historySummary' ∈ HashDomain
    (3)1. LL1NVRAM.historySummary' ∈ HashType
      BY (1)1, LL1SubtypeImplicationLemma DEF LL1SubtypeImplication
    (3)2. QED
      BY (3)1 DEF HashDomain
    (2)2. refStateHash' ∈ HashDomain
      BY (1)3 DEF HashDomain
    (2)3. QED
      BY (2)1, (2)2, HashTypeSafe
(1)5. QED
    BY (1)2, (1)3, (1)4
    DEF refPrivateStateEnc, refStateHash
### 4.3 Proof of Type Safety of the Memoir-Basic Spec

This module proves the type safety of the Memoir-Basic spec.

**EXTENDS** *MemoirLL1TypeLemmas*

**THEOREM** *LL1TypeSafe*  \(\triangleq\)  *LL1Spec*  \(\Rightarrow\)  *\(\square\)LL1TypeInvariant*

The top level of the proof is boilerplate TLA+ for an *Inv1*-style proof. First, we prove that the initial state satisfies *LL1TypeInvariant*. Second, we prove that the *LL1Next* predicate inductively preserves *LL1TypeInvariant*. Third, we use temporal induction to prove that these two conditions satisfy type safety over all behaviors.

\[1\] **LL1Init**  \(\Rightarrow\)  *LL1TypeInvariant*

The base case follows directly from the definition of *LL1Init*. There are a bunch of steps, but they are simple expansions of definitions and appeals to the type safety of the initial definitions.

\[2\] **1.** **HOLE**

\[2\] **2.** **PICK**  \(\text{symmetricKey} \in \text{SymmetricKeyType: LL1Init!}(\text{symmetricKey})!1\)

BY \[2\] **1**  **DEF**  *LL1Init*

\[2\] **2.** *initialPrivateStateEnc*  \(\triangleq\)  *SymmetricEncrypt*(\text{symmetricKey}, \text{InitialPrivateState})

\[2\] **2.** *initialStateHash*  \(\triangleq\)  *Hash*(\text{InitialPublicState}, \text{initialPrivateStateEnc})

\[2\] **2.** *initialHistoryStateBinding*  \(\triangleq\)  *Hash*(\text{BaseHashValue}, \text{initialStateHash})

\[2\] **2.** *initialAuthenticator*  \(\triangleq\)  *GenerateMAC*(\text{symmetricKey}, \text{initialHistoryStateBinding})

\[2\] **2.** *initialUntrustedStorage*  \(\triangleq\)  \[\text{publicStateure InitialPublicState, }\]

\[2\] **2.** \(\text{privateStateEnc} \mapsto \text{initialPrivateStateEnc, }\)

\[2\] **2.** \(\text{historySummary} \mapsto \text{BaseHashValue, }\)

\[2\] **2.** \(\text{authenticator} \mapsto \text{initialAuthenticator}\]

\[2\] **2.** *initialTrustedStorage*  \(\triangleq\)  \[\text{historySummary} \mapsto \text{BaseHashValue, }\]

\[2\] **2.** \(\text{symmetricKey} \mapsto \text{symmetricKey}\]

\[2\] **2.** \(\land\)  *initialPrivateStateEnc*  \(\in\)  *PrivateStateEncType*

\[2\] **2.** \(\land\)  *initialStateHash*  \(\in\)  *HashType*

\[2\] **2.** \(\land\)  *initialHistoryStateBinding*  \(\in\)  *HashType*

\[2\] **2.** \(\land\)  *initialAuthenticator*  \(\in\)  *MACType*

\[2\] **2.** \(\land\)  *initialUntrustedStorage*  \(\in\)  *LL1UntrustedStorageType*

\[2\] **2.** \(\land\)  *initialTrustedStorage*  \(\in\)  *LL1TrustedStorageType*

\[2\] **2.** *symmetricKey*  \(\in\)  *SymmetricKeyType*

BY \[2\] **2**

\[2\] **2.** **QED**

BY \[2\] **1**,  **LL1InitDefsTypeSafeLemma**

\[2\] **2.** **HIDE**  **DEF**  *initialPrivateStateEnc, initialStateHash, initialAuthenticator, initialUntrustedStorage, initialTrustedStorage*

\[2\] **2.** **LL1AvailableInputs**  \(\subseteq\)  *InputType*

\[2\] **2.** *LL1AvailableInputs*  \(\triangleq\)  *InitialAvailableInputs*

BY \[2\] **2**

\[2\] **2.** **QED**

BY \[2\] **1**,  \(\langle\text{3}\rangle\) **2**

\[2\] **2.** **LL1ObservedOutputs**  \(\subseteq\)  *OutputType*

\[2\] **2.** *LL1ObservedOutputs*  \(\triangleq\)  \{\}

BY \[2\] **2**

\[2\] **2.** **QED**

BY \[2\] **1**
2.6. $LL_1$ObservedAuthenticators $\subseteq MACType$

3.1. $LL_1$ObservedAuthenticators $=$ \{initialAuthenticator\}\n
   BY (2)2
   DEF initialAuthenticator, initialHistoryStateBinding, initialStateHash, initialPrivateStateEnc

3.2. initialAuthenticator $\in$ MACType

   BY (2)3
   (3)3. QED
   BY (3)1, (3)2

2.7. $LL_1$Disk $\in LL_1$UntrustedStorageType

3.1. $LL_1$Disk $=$ initialUntrustedStorage

   BY (2)2
   DEF initialUntrustedStorage, initialAuthenticator, initialHistoryStateBinding, initialStateHash, initialPrivateStateEnc

3.2. initialUntrustedStorage $\in LL_1$UntrustedStorageType

   BY (2)3
   (3)3. QED
   BY (3)1, (3)2

2.8. $LL_1$RAM $\in LL_1$UntrustedStorageType

3.1. $LL_1$RAM $=$ initialUntrustedStorage

   BY (2)2
   DEF initialUntrustedStorage, initialAuthenticator, initialHistoryStateBinding, initialStateHash, initialPrivateStateEnc

3.2. initialUntrustedStorage $\in LL_1$UntrustedStorageType

   BY (2)3
   (3)3. QED
   BY (3)1, (3)2

2.9. $LL_1$NVRAM $\in LL_1$TrustedStorageType

3.1. $LL_1$NVRAM $=$ initialTrustedStorage

   BY (2)2 DEF initialTrustedStorage

3.2. initialTrustedStorage $\in LL_1$TrustedStorageType

   BY (2)3
   (3)3. QED
   BY (3)1, (3)2

2.10. QED

BY (2)4, (2)5, (2)6, (2)7, (2)8, (2)9 DEF $LL_1$TypeInvariant

1.2. $LL_1$TypeInvariant $\land [LL_1\text{Next}\{LL_1\text{Vars}\}]$ $\Rightarrow LL_1$TypeInvariant′ 

The induction step is also straightforward. We assume the antecedents of the implication, then show that the consequent holds for all eight $LL_1$Next actions plus stuttering.

2.1. HAVE $LL_1$TypeInvariant $\land [LL_1\text{Next}\{LL_1\text{Vars}\}$

2.2. CASE UNCHANGED $LL_1$Vars

Type safety is inductively trivial for a stuttering step.

3.1. $LL_1$AvailableInputs′ $\subseteq InputType$

   (4)1. $LL_1$AvailableInputs $\subseteq InputType$

   BY (2)1 DEF $LL_1$TypeInvariant

   (4)2. UNCHANGED $LL_1$AvailableInputs

   BY (2)2 DEF $LL_1$Vars

   (4)3. QED

   BY (4)1, (4)2

3.2. $LL_1$ObservedOutputs′ $\subseteq OutputType$

   (4)1. $LL_1$ObservedOutputs $\subseteq OutputType$

   BY (2)1 DEF $LL_1$TypeInvariant

70
(4) 2. UNCHANGED $LL1ObservedOutputs$
    BY (2) 2 DEF $LL1Vars$
(4) 3. QED
    BY (4) 1, (4) 2

(3) 3. $LL1ObservedAuthenticators' \subseteq MACType$
    (4) 1. $LL1ObservedAuthenticators \subseteq MACType$
        BY (2) 1 DEF $LL1TypeInvariant$
    (4) 2. UNCHANGED $LL1ObservedAuthenticators$
        BY (2) 2 DEF $LL1Vars$
    (4) 3. QED
        BY (4) 1, (4) 2

(3) 4. $LL1Disk' \in LL1UntrustedStorageType$
    (4) 1. $LL1Disk \in LL1UntrustedStorageType$
        BY (2) 1 DEF $LL1TypeInvariant$
    (4) 2. UNCHANGED $LL1Disk$
        BY (2) 2 DEF $LL1Vars$
    (4) 3. QED
        BY (4) 1, (4) 2

(3) 5. $LL1RAM' \in LL1UntrustedStorageType$
    (4) 1. $LL1RAM \in LL1UntrustedStorageType$
        BY (2) 1 DEF $LL1TypeInvariant$
    (4) 2. UNCHANGED $LL1RAM$
        BY (2) 2 DEF $LL1Vars$
    (4) 3. QED
        BY (4) 1, (4) 2

(3) 6. $LL1NVRAM' \in LL1TrustedStorageType$
    (4) 1. $LL1NVRAM \in LL1TrustedStorageType$
        BY (2) 1 DEF $LL1TypeInvariant$
    (4) 2. UNCHANGED $LL1NVRAM$
        BY (2) 2 DEF $LL1Vars$
    (4) 3. QED
        BY (4) 1, (4) 2

(3) 7. QED
    BY (3) 1, (3) 2, (3) 3, (3) 4, (3) 5, (3) 6 DEF $LL1TypeInvariant$

(2) 3. CASE $LL1Next$
(3) 1. CASE $LL1MakeInputAvailable$

Type safety is straightforward for a $LL1MakeInputAvailable$ action.

(4) 1. PICK $input \in InputType : LL1MakeInputAvailable!(input)$
    BY (3) 1 DEF $LL1MakeInputAvailable$

(4) 2. $LL1AvailableInputs' \subseteq InputType$
    (5) 1. $LL1AvailableInputs \subseteq InputType$
        BY (2) 1 DEF $LL1TypeInvariant$
    (5) 2. $LL1AvailableInputs' = LL1AvailableInputs \cup \{input\}$
        BY (4) 1
    (5) 3. $input \in InputType$
        BY (4) 1
    (5) 4. QED
        BY (5) 1, (5) 2, (5) 3

(4) 3. $LL1ObservedOutputs' \subseteq OutputType$
    (5) 1. $LL1ObservedOutputs \subseteq OutputType$
        BY (2) 1 DEF $LL1TypeInvariant$
    (5) 2. UNCHANGED $LL1ObservedOutputs$
BY (4)1
(5)3. QED
BY (5)1, (5)2
(4)4. $LL1^\text{ObservedAuthenticators}' \subseteq \text{MACType}$
(5)1. $LL1^\text{ObservedAuthenticators} \subseteq \text{MACType}$
BY (2)1 DEF $LL1^\text{TypeInvariant}$
(5)2. UNCHANGED $LL1^\text{ObservedAuthenticators}$
BY (4)1
(5)3. QED
BY (5)1, (5)2
(4)5. $LL1^\text{Disk}' \in LL1^\text{UntrustedStorageType}$
(5)1. $LL1^\text{Disk} \in LL1^\text{UntrustedStorageType}$
BY (2)1 DEF $LL1^\text{TypeInvariant}$
(5)2. UNCHANGED $LL1^\text{Disk}$
BY (4)1
(5)3. QED
BY (5)1, (5)2
(4)6. $LL1^\text{RAM}' \in LL1^\text{UntrustedStorageType}$
(5)1. $LL1^\text{RAM} \in LL1^\text{UntrustedStorageType}$
BY (2)1 DEF $LL1^\text{TypeInvariant}$
(5)2. UNCHANGED $LL1^\text{RAM}$
BY (4)1
(5)3. QED
BY (5)1, (5)2
(4)7. $LL1^\text{NVRAM}' \in LL1^\text{TrustedStorageType}$
(5)1. $LL1^\text{NVRAM} \in LL1^\text{TrustedStorageType}$
BY (2)1 DEF $LL1^\text{TypeInvariant}$
(5)2. UNCHANGED $LL1^\text{NVRAM}$
BY (4)1
(5)3. QED
BY (5)1, (5)2
(4)8. QED
BY (4)2, (4)3, (4)4, (4)5, (4)6, (4)7 DEF $LL1^\text{TypeInvariant}$
(3)2. CASE $LL1^\text{PerformOperation}$

For a $LL1^\text{PerformOperation}$ action, we just walk through the definitions. Type safety follows directly.

(4)1. PICK $input \in LL1^\text{AvailableInputs} : LL1^\text{PerformOperation}!(input)!1$
BY (3)2 DEF $LL1^\text{PerformOperation}$
(4) state$\text{Hash} \triangleq \text{Hash}(LL1^\text{RAM}.\text{publicState}, LL1^\text{RAM}.\text{privateStateEnc})$
(4) history$\text{StateBinding} \triangleq \text{Hash}(LL1^\text{RAM}.\text{historySummary}, \text{stateHash})$
(4) privateState $\triangleq \text{SymmetricDecrypt}(LL1^\text{NVRAM}.\text{symmetricKey}, LL1^\text{RAM}.\text{privateStateEnc})$
(4) sResult $\triangleq \text{Service}(LL1^\text{RAM}.\text{publicState}, \text{privateState}, \text{input})$
(4) newPrivateStateEnc $\triangleq$
SymmetricEncrypt($LL1^\text{NVRAM}.\text{symmetricKey}, \text{sResult}.\text{newPrivateState}$)
(4) newHistorySummary $\triangleq \text{Hash}(LL1^\text{NVRAM}.\text{historySummary}, \text{input})$
(4) new$\text{StateHash} \triangleq \text{Hash}(\text{sResult}.\text{newPublicState}, \text{newPrivateStateEnc})$
(4) newHistoryStateBinding $\triangleq \text{Hash}(\text{newHistorySummary}, \text{newStateHash})$
(4) new$\text{Authenticator} \triangleq \text{GenerateMAC}(LL1^\text{NVRAM}.\text{symmetricKey}, \text{newHistoryStateBinding})$
(4)2. $\land \text{stateHash} \in \text{HashType}$
$\land \text{privateState} \in \text{PrivateStateType}$
$\land \text{sResult} \in \text{ServiceResultType}$
$\land \text{sResult}.\text{newPublicState} \in \text{PublicStateType}$
∧ sResult.newPrivateState ∈ PrivateStateType
∧ sResult.output ∈ OutputType
∧ newPrivateStateEnc ∈ PrivateStateEncType
∧ newHistorySummary ∈ HashType
∧ newStateHash ∈ HashType
∧ newHistoryStateBinding ∈ HashType
∧ newAuthenticator ∈ MACType
(5.1) input ∈ LL1AvailableInputs
BY (4.1)
(5.2) LL1TypeInvariant
BY (2.1)
(5.3) QED
BY (5.1), (5.2), LL1PerformOperationDefsTypeSafeLemma
(4) HIDE DEF stateHash, historyStateBinding, privateState, sResult, newPrivateStateEnc, 
newHistorySummary, newStateHash, newHistoryStateBinding, newAuthenticator
(4.3) LL1AvailableInputs' ⊆ InputType
(5.1) LL1AvailableInputs ⊆ InputType
BY (2.1) DEF LL1TypeInvariant
(5.2) UNCHANGED LL1AvailableInputs
BY (4.1)
(5.3) QED
BY (5.1), (5.2)
(4.4) LL1ObservedOutputs' ⊆ OutputType
(5.1) LL1ObservedOutputs ⊆ OutputType
BY (2.1) DEF LL1TypeInvariant
(5.2) LL1ObservedOutputs' = LL1ObservedOutputs ∪ {sResult.output}
BY (4.1) DEF sResult, privateState
(5.3) sResult.output ∈ OutputType
BY (4.2)
(5.4) QED
BY (5.1), (5.2), (5.3)
(4.5) LL1ObservedAuthenticators' ⊆ MACType
(5.1) LL1ObservedAuthenticators ⊆ MACType
BY (2.1) DEF LL1TypeInvariant
(5.2) LL1ObservedAuthenticators' = 
LL1ObservedAuthenticators ∪ {newAuthenticator}
BY (4.1) DEF newAuthenticator, newHistoryStateBinding, newStateHash, 
newHistorySummary, newPrivateStateEnc, sResult, privateState
(5.3) newAuthenticator ∈ MACType
BY (4.2)
(5.4) QED
BY (5.1), (5.2), (5.3)
(4.6) LL1Disk' ∈ LL1UntrustedStorageType
(5.1) LL1Disk ∈ LL1UntrustedStorageType
BY (2.1) DEF LL1TypeInvariant
(5.2) UNCHANGED LL1Disk
BY (4.1)
(5.3) QED
BY (5.1), (5.2)
(4.7) LL1RAM' ∈ LL1UntrustedStorageType
(5.1) LL1RAM' = [publicState → sResult.newPublicState, 
privateStateEnc → newPrivateStateEnc,
For a LL1RepeatOperation action, we just walk through the definitions. Type safety follows directly.

(4.1) \text{pick} \ input \in \text{LL1AvailableInputs} : \text{LL1RepeatOperation!}((\text{input})!1

(4.2) stateHash \triangleq \text{Hash}(\text{LL1RAM.publicState}, \text{LL1RAM.privateStateEnc})

(4.3) historyStateBinding \triangleq \text{Hash}(\text{LL1RAM.historySummary}, \text{stateHash})

(4.4) privateKey \triangleq \text{SymmetricDecrypt}((\text{LL1NRAM.symmetricKey}, \text{LL1RAM.privateStateEnc})

(4.5) sResult \triangleq \text{Service}(\text{LL1RAM.publicState}, \text{privateState}, \text{input})

(4.6) newPrivateStateEnc \triangleq \text{SymmetricEncrypt}(\text{LL1NRAM.symmetricKey}, \text{sResult.newPrivateState})

(4.7) newStateHash \triangleq \text{Hash}(\text{sResult.newPublicState}, \text{newPrivateStateEnc})

(4.8) newHistoryStateBinding \triangleq \text{Hash}(\text{LL1NRAM.historySummary}, \text{newStateHash})

(4.9) newAuthenticator \triangleq \text{GenerateMAC}(\text{LL1NRAM.symmetricKey}, \text{newHistoryStateBinding})

(4.2) \quad \land \quad \text{stateHash} \in \text{HashType}
\quad \land \quad \text{privateState} \in \text{PrivateStateType}
\quad \land \quad \text{sResult} \in \text{ServiceResultType}
\quad \land \quad \text{sResult.newPublicState} \in \text{PublicStateType}
\quad \land \quad \text{sResult.newPrivateState} \in \text{PrivateStateType}
\quad \land \quad \text{sResult.output} \in \text{OutputType}
\quad \land \quad \text{newPrivateStateEnc} \in \text{PrivateStateEncType}
\quad \land \quad \text{newStateHash} \in \text{HashType}
\quad \land \quad \text{newHistoryStateBinding} \in \text{HashType}
\quad \land \quad \text{newAuthenticator} \in \text{MACType}

(5.1) \text{input} \in \text{LL1AvailableInputs}
(5.2) \textit{LL1TypeInvariant}
BY (2.1)
(5.3) QED
BY (5.1), (5.2), \textit{LL1RepeatOperationDefsTypeSafeLemma}

(4) HIDE DEF \textit{stateHash}, \textit{historyStateBinding}, \textit{privateState}, \textit{sResult}, \textit{newPrivateStateEnc},
newStateHash, newHistoryStateBinding, newAuthenticator

(4.3) \textit{LL1AvailableInputs}' \subseteq \textit{InputType}
(5.1) \textit{LL1AvailableInputs} \subseteq \textit{InputType}
BY (2.1) DEF \textit{LL1TypeInvariant}
(5.2) UNCHANGED \textit{LL1AvailableInputs}
BY (4.1)
(5.3) QED
BY (5.1), (5.2)

(4.4) \textit{LL1ObservedOutputs}' \subseteq \textit{OutputType}
(5.1) \textit{LL1ObservedOutputs} \subseteq \textit{OutputType}
BY (2.1) DEF \textit{LL1TypeInvariant}
(5.2) \textit{LL1ObservedOutputs}' = \textit{LL1ObservedOutputs} \cup \{\textit{sResult}.output\}
BY (4.1) DEF \textit{sResult}, \textit{privateState}
(5.3) \textit{sResult}.output \in \textit{OutputType}
BY (4.2)
(5.4) QED
BY (5.1), (5.2), (5.3)

(4.5) \textit{LL1ObservedAuthenticators}' \subseteq \textit{MACType}
(5.1) \textit{LL1ObservedAuthenticators} \subseteq \textit{MACType}
BY (2.1) DEF \textit{LL1TypeInvariant}
(5.2) \textit{LL1ObservedAuthenticators}' = \\
\textit{LL1ObservedAuthenticators} \cup \{\textit{newAuthenticator}\}
BY (4.1) DEF \textit{newAuthenticator}, \textit{newHistoryStateBinding}, \textit{newStateHash},
\textit{newPrivateStateEnc}, \textit{sResult}, \textit{privateState}
(5.3) \textit{newAuthenticator} \in \textit{MACType}
BY (4.2)
(5.4) QED
BY (5.1), (5.2), (5.3)

(4.6) \textit{LL1Disk}' \in \textit{LL1UntrustedStorageType}
(5.1) \textit{LL1Disk} \in \textit{LL1UntrustedStorageType}
BY (2.1) DEF \textit{LL1TypeInvariant}
(5.2) UNCHANGED \textit{LL1Disk}
BY (4.1)
(5.3) QED
BY (5.1), (5.2)

(4.7) \textit{LL1RAM}' \in \textit{LL1UntrustedStorageType}
(5.1) \textit{LL1RAM}' = [\textit{publicState} \mapsto \textit{sResult}.newPublicState, \\
\textit{privateStateEnc} \mapsto \textit{newPrivateStateEnc}, \\
\textit{historySummary} \mapsto \textit{LL1NVRAM}.historySummary, \\
\textit{authenticator} \mapsto \textit{newAuthenticator}]
BY (4.1) DEF \textit{newAuthenticator}, \textit{newHistoryStateBinding}, \textit{newStateHash},
\textit{newPrivateStateEnc}, \textit{sResult}, \textit{privateState}
(5.2) \textit{sResult}.newPublicState \in \textit{PublicStateType}
BY (4.2)
(5.3) \textit{newPrivateStateEnc} \in \textit{PrivateStateEncType}
BY (4.2)
(5.4) \textit{LL1NVRAM}.historySummary \in \textit{HashType}
Type safety is straightforward for a `LL1Restart` action.

(3.4) **CASE LL1Restart**

(4.1) **Pick** `untrustedStorage ∈ LL1UntrustedStorageType`, `randomSymmetricKey ∈ SymmetricKeyType \ {LL1NVRAM . symmetricKey}`, `hash ∈ HashType`:

`LL1Restart!(untrustedStorage, randomSymmetricKey, hash)`

(4.2) `LL1AvailableInputs' ⊆ InputType`

(5.1) `LL1AvailableInputs ⊆ InputType`

(5.2) **UNCHANGED LL1AvailableInputs**

(5.3) **QED**

(4.3) `LL1ObservedOutputs' ⊆ OutputType`

(5.1) `LL1ObservedOutputs ⊆ OutputType`

(5.2) **UNCHANGED LL1ObservedOutputs**

(5.3) **QED**

(4.4) `LL1ObservedAuthenticators' ⊆ MACType`

(5.1) `LL1ObservedAuthenticators ⊆ MACType`

(5.2) **UNCHANGED LL1ObservedAuthenticators**

(5.3) **QED**

(4.5) `LL1Disk' ∈ LL1UntrustedStorageType`

(5.1) `LL1Disk ∈ LL1UntrustedStorageType`

(5.2) **UNCHANGED LL1Disk**

(5.3) **QED**

(4.6) `LL1RAM' ∈ LL1UntrustedStorageType`

(5.1) `untrustedStorage ∈ LL1UntrustedStorageType`

(5.2) **QED**
(5) 2. $LL1RAM' = untrustedStorage$
   BY (4) 1
(5) 3. QED
   BY (5) 1, (5) 2
(4) 7. $LL1NVRAM' \in LL1TrustedStorageType$
   (5) 1. $LL1NVRAM \in LL1TrustedStorageType$
      BY (2) 1 DEF $LL1TypeInvariant$
   (5) 2. UNCHANGED $LL1NVRAM$
      BY (4) 1
   (5) 3. QED
      BY (5) 1, (5) 2
(4) 8. QED
      BY (4) 2, (4) 3, (4) 4, (4) 5, (4) 6, (4) 7 DEF $LL1TypeInvariant$
(3) 5. CASE $LL1ReadDisk$

Type safety is straightforward for a $LL1ReadDisk$ action.

(4) 1. $LL1AvailableInputs' \subseteq InputType$
   (5) 1. $LL1AvailableInputs \subseteq InputType$
      BY (2) 1 DEF $LL1TypeInvariant$
   (5) 2. UNCHANGED $LL1AvailableInputs$
      BY (3) 5 DEF $LL1ReadDisk$
   (5) 3. QED
      BY (5) 1, (5) 2
(4) 2. $LL1ObservedOutputs' \subseteq OutputType$
   (5) 1. $LL1ObservedOutputs \subseteq OutputType$
      BY (2) 1 DEF $LL1TypeInvariant$
   (5) 2. UNCHANGED $LL1ObservedOutputs$
      BY (3) 5 DEF $LL1ReadDisk$
   (5) 3. QED
      BY (5) 1, (5) 2
(4) 3. $LL1ObservedAuthenticators' \subseteq MACType$
   (5) 1. $LL1ObservedAuthenticators \subseteq MACType$
      BY (2) 1 DEF $LL1TypeInvariant$
   (5) 2. UNCHANGED $LL1ObservedAuthenticators$
      BY (3) 5 DEF $LL1ReadDisk$
   (5) 3. QED
      BY (5) 1, (5) 2
(4) 4. $LL1Disk' \in LL1UntrustedStorageType$
   (5) 1. $LL1Disk \in LL1UntrustedStorageType$
      BY (2) 1 DEF $LL1TypeInvariant$
   (5) 2. UNCHANGED $LL1Disk$
      BY (3) 5 DEF $LL1ReadDisk$
   (5) 3. QED
      BY (5) 1, (5) 2
(4) 5. $LL1RAM' \in LL1UntrustedStorageType$
   (5) 1. $LL1Disk \in LL1UntrustedStorageType$
      BY (2) 1 DEF $LL1TypeInvariant$
   (5) 2. $LL1RAM' = LL1Disk$
      BY (3) 5 DEF $LL1ReadDisk$
   (5) 3. QED
      BY (5) 1, (5) 2
(4) 6. $LL1NVRAM' \in LL1TrustedStorageType$
   (5) 1. $LL1NVRAM \in LL1TrustedStorageType$

77
Type safety is straightforward for a LL1WriteDisk action.

(4.1) LL1AvailableInputs' ⊆ InputType
(5.1) LL1AvailableInputs ⊆ InputType
   BY (2) 1 DEF LL1TypeInvariant
(5.2) UNCHANGED LL1AvailableInputs
   BY (3) 6 DEF LL1WriteDisk
(5.3) QED
   BY (5) 1, (5) 2

(4.2) LL1ObservedOutputs' ⊆ OutputType
(5.1) LL1ObservedOutputs ⊆ OutputType
   BY (2) 1 DEF LL1TypeInvariant
(5.2) UNCHANGED LL1ObservedOutputs
   BY (3) 6 DEF LL1WriteDisk
(5.3) QED
   BY (5) 1, (5) 2

(4.3) LL1ObservedAuthenticators' ⊆ MACType
(5.1) LL1ObservedAuthenticators ⊆ MACType
   BY (2) 1 DEF LL1TypeInvariant
(5.2) UNCHANGED LL1ObservedAuthenticators
   BY (3) 6 DEF LL1WriteDisk
(5.3) QED
   BY (5) 1, (5) 2

(4.4) LL1Disk' ∈ LL1UntrustedStorageType
(5.1) LL1RAM ∈ LL1UntrustedStorageType
   BY (2) 1 DEF LL1TypeInvariant
(5.2) LL1Disk' = LL1RAM
   BY (3) 6 DEF LL1WriteDisk
(5.3) QED
   BY (5) 1, (5) 2

(4.5) LL1RAM' ∈ LL1UntrustedStorageType
(5.1) LL1RAM ∈ LL1UntrustedStorageType
   BY (2) 1 DEF LL1TypeInvariant
(5.2) UNCHANGED LL1RAM
   BY (3) 6 DEF LL1WriteDisk
(5.3) QED
   BY (5) 1, (5) 2

(4.6) LL1NVRAM' ∈ LL1TrustedStorageType
(5.1) LL1NVRAM ∈ LL1TrustedStorageType
   BY (2) 1 DEF LL1TypeInvariant
(5.2) UNCHANGED LL1NVRAM
   BY (3) 6 DEF LL1WriteDisk
(5.3) QED
   BY (5) 1, (5) 2

(4.7) QED
Type safety is straightforward for a $LL1CorruptRAM$ action.

(4.1) PICK $untrustedStorage \in LL1UntrustedStorageType$,
      $fakeSymmetricKey \in SymmetricKeyType \setminus \{LL1NVRAM.symmetricKey\}$,
      $hash \in HashType$ :
      $LL1CorruptRAM!(untrustedStorage, fakeSymmetricKey, hash)$

BY (3) $LL1CorruptRAM$

(4.2) $LL1AvailableInputs' \subseteq InputType$

(5.1) $LL1AvailableInputs \subseteq InputType$

BY (2) $LL1TypeInvariant$

(5.2) UNCHANGED $LL1AvailableInputs$

BY (4) $LL1TypeInvariant$

(5.3) QED

BY (5.1), (5.2)

(4.3) $LL1ObservedOutputs' \subseteq OutputType$

(5.1) $LL1ObservedOutputs \subseteq OutputType$

BY (2) $LL1TypeInvariant$

(5.2) UNCHANGED $LL1ObservedOutputs$

BY (4) $LL1TypeInvariant$

(5.3) QED

BY (5.1), (5.2)

(4.4) $LL1ObservedAuthenticators' \subseteq MACType$

(5.1) $LL1ObservedAuthenticators \subseteq MACType$

BY (2) $LL1TypeInvariant$

(5.2) UNCHANGED $LL1ObservedAuthenticators$

BY (4) $LL1TypeInvariant$

(5.3) QED

BY (5.1), (5.2)

(4.5) $LL1Disk' \in LL1UntrustedStorageType$

(5.1) $LL1Disk \in LL1UntrustedStorageType$

BY (2) $LL1TypeInvariant$

(5.2) UNCHANGED $LL1Disk$

BY (4) $LL1TypeInvariant$

(5.3) QED

BY (5.1), (5.2)

(4.6) $LL1RAM' \in LL1UntrustedStorageType$

(5.1) $untrustedStorage \in LL1UntrustedStorageType$

BY (4) $LL1TypeInvariant$

(5.2) $LL1RAM' = untrustedStorage$

BY (4) $LL1TypeInvariant$

(5.3) QED

BY (5.1), (5.2)

(4.7) $LL1NVRAM' \in LL1TrustedStorageType$

(5.1) $LL1NVRAM \in LL1TrustedStorageType$

BY (2) $LL1TypeInvariant$

(5.2) UNCHANGED $LL1NVRAM$

BY (4) $LL1TypeInvariant$

(5.3) QED

BY (5.1), (5.2)

(4.8) QED

BY (4) $LL1TypeInvariant$
(3)8. CASE \textit{LL1\text{RestrictedCorruption}}

Type safety is straightforward for a \textit{LL1\text{RestrictedCorruption}} action.

(4)2. \textit{LL1\text{AvailableInputs}'} \subseteq \textit{InputType}

(5)1. \textit{LL1\text{AvailableInputs} } \subseteq \textit{InputType}

BY (2)1 DEF \textit{LL1\text{TypeInvariant}}

(5)2. UNCHANGED \textit{LL1\text{AvailableInputs}}

BY (3)8 DEF \textit{LL1\text{RestrictedCorruption}}

(5)3. QED

BY (5)1, (5)2

(4)3. \textit{LL1\text{ObservedOutputs}'} \subseteq \textit{OutputType}

(5)1. \textit{LL1\text{ObservedOutputs} } \subseteq \textit{OutputType}

BY (2)1 DEF \textit{LL1\text{TypeInvariant}}

(5)2. UNCHANGED \textit{LL1\text{ObservedOutputs}}

BY (3)8 DEF \textit{LL1\text{RestrictedCorruption}}

(5)3. QED

BY (5)1, (5)2

(4)4. \textit{LL1\text{ObservedAuthenticators}'} \subseteq \textit{MACType}

(5)1. \textit{LL1\text{ObservedAuthenticators} } \subseteq \textit{MACType}

BY (2)1 DEF \textit{LL1\text{TypeInvariant}}

(5)2. UNCHANGED \textit{LL1\text{ObservedAuthenticators}}

BY (3)8 DEF \textit{LL1\text{RestrictedCorruption}}

(5)3. QED

BY (5)1, (5)2

(4)5. \textit{LL1\text{Disk}'} \in \textit{LL1\text{UntrustedStorageType}}

(5)1. \textit{LL1\text{Disk} } \in \textit{LL1\text{UntrustedStorageType}}

BY (2)1 DEF \textit{LL1\text{TypeInvariant}}

(5)2. UNCHANGED \textit{LL1\text{Disk}}

BY (3)8 DEF \textit{LL1\text{RestrictedCorruption}}

(5)3. QED

BY (5)1, (5)2

(4)6. \textit{LL1\text{RAM}'} \in \textit{LL1\text{UntrustedStorageType}}

(5)1. CASE \textit{LL1\text{RestrictedCorruption}!ram!	ext{unchanged}}

(6)1. \textit{LL1\text{RAM} } \in \textit{LL1\text{UntrustedStorageType}}

BY (2)1 DEF \textit{LL1\text{TypeInvariant}}

(6)2. UNCHANGED \textit{LL1\text{RAM}}

BY (5)1

(6)3. QED

BY (6)1, (6)2

(5)2. CASE \textit{LL1\text{RestrictedCorruption}!ram!	ext{trashed}}

(6)1. PICK \textit{untrustedStorage } \in \textit{LL1\text{UntrustedStorageType}},

\textit{randomSymmetricKey } \in \textit{SymmetricKeyType \setminus \{LL1\text{NVRAM}.\text{symmetricKey}\}},

\textit{hash } \in \textit{HashType}:

\textit{LL1\text{RestrictedCorruption}!ram!	ext{trashed}!}

\texttt{\{untrustedStorage, randomSymmetricKey, hash\}}

BY (5)2

(6)2. \textit{untrustedStorage } \in \textit{LL1\text{UntrustedStorageType}}

BY (6)1

(6)3. \textit{LL1\text{RAM}'} = \textit{untrustedStorage}

BY (6)1

(6)4. QED

BY (6)2, (6)3

(5)3. QED
BY $\langle 3 \rangle 8$, $\langle 5 \rangle 1$, $\langle 5 \rangle 2$ DEF $LL1$\textit{RestrictedCorruption}

(4.7) $LL1$NV\textit{RAM}' $\in LL1$\textit{TrustedStorageType}

(5.1) PICK garbageHistorySummary $\in$ HashType :

\[
\text{LL1RestrictedCorruption!nvram!(garbageHistorySummary)}
\]

BY $\langle 3 \rangle 8$ DEF $LL1$\textit{RestrictedCorruption}

(5.2) garbageHistorySummary $\in$ HashType

BY $\langle 5 \rangle 1$

(5.3) $LL1$NV\textit{RAM}.symmetricKey $\in$ SymmetricKey\textit{Type}

BY $\langle 2 \rangle 1$, $LL1$\textit{SubtypeImplicationLemma}$DEF$ $LL1$\textit{SubtypeImplication}

(5.4) $LL1$NV\textit{RAM}' = [historySummary $\mapsto$ garbageHistorySummary, symmetricKey $\mapsto$ $LL1$NV\textit{RAM}.symmetricKey]

BY $\langle 5 \rangle 1$

(5.5) QED

BY $\langle 5 \rangle 2$, $\langle 5 \rangle 3$, $\langle 5 \rangle 4$ DEF $LL1$\textit{TrustedStorageType}

(4.8) QED

BY $\langle 4 \rangle 2$, $\langle 4 \rangle 3$, $\langle 4 \rangle 4$, $\langle 4 \rangle 5$, $\langle 4 \rangle 6$, $\langle 4 \rangle 7$ DEF $LL1$\textit{TypeInvariant}

(3.9) QED

BY $\langle 2 \rangle 3$, $\langle 3 \rangle 1$, $\langle 3 \rangle 2$, $\langle 3 \rangle 3$, $\langle 3 \rangle 4$, $\langle 3 \rangle 5$, $\langle 3 \rangle 6$, $\langle 3 \rangle 7$, $\langle 3 \rangle 8$ DEF $LL1$\textit{Next}

(2.4) QED

BY $\langle 2 \rangle 1$, $\langle 2 \rangle 2$, $\langle 2 \rangle 3$

$\langle 1 \rangle 3$. QED

Using the $\text{Inv1}$ proof rule, the base case and the induction step together imply that the invariant always holds.

(2.1) $LL1$\textit{TypeInvariant} $\land \Box[LL1$\textit{Next}]_LL1\textit{Vars}$ $\Rightarrow \Box LL1$\textit{TypeInvariant}$

BY $\langle 1 \rangle 2$, $\text{Inv1}$

(2.2) QED

BY $\langle 1 \rangle 1$, $\langle 2 \rangle 1$ DEF $LL1$\textit{Spec}
This module states and proves several lemmas that are useful for proving the Memoir-Basic invariance properties.

The lemmas in this module are:
- SymmetricKeyConstantLemma
- LL1NVRAHistorySummaryUncorruptedUnchangedLemma
- LL1RepeatOperationUnchangedObservedOutputsLemma
- LL1RepeatOperationUnchangedAuthenticatedHistoryStateBindingsLemma
- LL1RAMUnforgeabilityUnchangedLemma
- LL1DiskUnforgeabilityUnchangedLemma
- InclusionUnchangedLemma
- CardinalityUnchangedLemma
- UniquenessUnchangedLemma
- UnchangedAuthenticatedHistoryStateBindingsLemma

Proof relating to cardinality require some basic properties of inequalities on natural numbers. The prover requires that we state these explicitly.

**THEOREM LEQTransitive**

\( \forall n, m, q \in \text{Nat} : n \leq m \land m \leq q \Rightarrow n \leq q \)

**OBVIOUS**

\{ by isabelle "(auto dest : nat_leq_trans)" \}

**THEOREM GEQorLT**

\( \forall n, m \in \text{Nat} : n \geq m \equiv \neg (m > n) \)

**OBVIOUS**

\{ by isabelle "(auto simp : nat_not_less dest : nat_leq_less_trans)" \}

The **SymmetricKeyConstantLemma** states that the LL1Next actions do not change the value of the symmetric key in NVRAM. The proof follows directly from the definition of the actions.

**THEOREM SymmetricKeyConstantLemma**

\( [\text{LL1Next}]_{\text{LL1Vars}} \Rightarrow \text{UNCHANGED} \text{LL1NVRA._symmetricKey} \)

\{1\}1. HAVE \( [\text{LL1Next}]_{\text{LL1Vars}} \)

\{1\}2. CASE UNCHANGED LL1Vars

\( \langle 2 \rangle \)1. UNCHANGED LL1NVRA

BY \( \langle 1 \rangle \)2 DEF LL1Vars

\( \langle 2 \rangle \)2. QED

BY \( \langle 2 \rangle \)1

\{1\}3. CASE LL1Next

\( \langle 2 \rangle \)1. CASE LL1MakeInputAvailable

\( \langle 3 \rangle \)1. UNCHANGED LL1NVRA

BY \( \langle 2 \rangle \)1 DEF LL1MakeInputAvailable

\( \langle 3 \rangle \)2. QED

BY \( \langle 3 \rangle \)1, HashCardinalityAccumulative

\( \langle 2 \rangle \)2. CASE LL1PerformOperation

\( \langle 3 \rangle \)1. PICK \( \text{input} \in \text{LL1AvailableInputs} : \text{LL1PerformOperation!}(\text{input})!1 \)

BY \( \langle 2 \rangle \)2 DEF LL1PerformOperation

\( \langle 3 \rangle \)2. LL1NVRA' = [historySummary \mapsto LL1PerformOperation! (input)!1 newHistorySummary, symmetricKey \mapsto LL1NVRA._symmetricKey]

BY \( \langle 3 \rangle \)1

\( \langle 3 \rangle \)3. LL1NVRA._symmetricKey' = LL1NVRA._symmetricKey

BY \( \langle 3 \rangle \)2

\( \langle 3 \rangle \)4. QED
by ⟨3⟩3

(2)3. CASE \textit{LL1RepeatOperation}
(3)1. UNCHANGED \textit{LL1NVRAM}
    by (2)3 DEF \textit{LL1RepeatOperation}
(3)2. QED
    by (3)1

(2)4. CASE \textit{LL1Restart}
(3)1. UNCHANGED \textit{LL1NVRAM}
    by (2)4 DEF \textit{LL1Restart}
(3)2. QED
    by (3)1

(2)5. CASE \textit{LL1ReadDisk}
(3)1. UNCHANGED \textit{LL1NVRAM}
    by (2)5 DEF \textit{LL1ReadDisk}
(3)2. QED
    by (3)1

(2)6. CASE \textit{LL1WriteDisk}
(3)1. UNCHANGED \textit{LL1NVRAM}
    by (2)6 DEF \textit{LL1WriteDisk}
(3)2. QED
    by (3)1

(2)7. CASE \textit{LL1CorruptRAM}
(3)1. UNCHANGED \textit{LL1NVRAM}
    by (2)7 DEF \textit{LL1CorruptRAM}
(3)2. QED
    by (3)1

(2)8. CASE \textit{LL1RestrictedCorruption}
(3)1. PICK \textit{garbageHistorySummary} ∈ \textit{HashType}:
    \textit{LL1RestrictedCorruption!nram!}(\textit{garbageHistorySummary})
    by (2)8 DEF \textit{LL1RestrictedCorruption}
(3)2. \textit{LL1NVRAM'} = [\textit{historySummary} → \textit{garbageHistorySummary},
    \textit{symmetricKey} → \textit{LL1NVRAM}.\textit{symmetricKey}]
    by (3)1
(3)3. QED
    by (3)2

(2)9. QED
    by (1)3, (2)1, (2)2, (2)3, (2)4, (2)5, (2)6, (2)7, (2)8 DEF \textit{LL1Next}
(1)4. QED
    by (1)1, (1)2, (1)3

The \textit{LL1NVRAMHistorySummaryUncorruptedUnchangedLemma} states that, if there are no changes to the \textit{NVRAM} or to the authentication status of any history state binding, then the truth value of the \textit{LL1NVRAMHistorySummaryUncorrupted} predicate is unchanged.

THEOREM \textit{LL1NVRAMHistorySummaryUncorruptedUnchangedLemma} \(\equiv\)

\((\land \text{LL1TypeInvariant}
    \land \text{UNCHANGED LL1NVRAM}
    \land \forall \text{historyStateBinding1} ∈ \textit{HashType}:
        \text{UNCHANGED LL1HistoryStateBindingAuthenticated(historyStateBinding1)})
\Rightarrow
\text{UNCHANGED LL1NVRAMHistorySummaryUncorrupted}\)
We begin by assuming the antecedent.

(1) \begin{align*}
1. \text{have} & \land \text{LL1 TypeInvariant} \\
& \land \text{UNCHANGED LL1 NVRAM} \\
& \land \forall \text{historyStateBinding1 }\in \text{HashType} : \\
& \text{UNCHANGED LL1 HistoryStateBindingAuthenticated(historyStateBinding1)}
\end{align*}

We first consider the case in which the truth value of the predicate is true.

(1)2. CASE LL1 NVRAM HistorySummary Uncorrupted = TRUE

To show that the value is unchanged, it suffices to show that the value is true in the primed state.

(2)1. SUFFICES

\begin{align*}
\text{ASSUME TRUE} \\
\text{PROVE LL1 NVRAM HistorySummary Uncorrupted'} & = \text{TRUE}
\end{align*}

by (1)2

We pick some state hash for which the LL1 NVRAM HistorySummary Uncorrupted is true in the unprimed state.

(2)2. PICK stateHash ∈ HashType :

\begin{align*}
\text{LL1 NVRAM HistorySummary Uncorrupted}'(stateHash) & = \text{TRUE}
\end{align*}

by (1)2

We copy the definition from the let in LL1 NVRAM HistorySummary Uncorrupted.

\begin{align*}
(2) \text{historyStateBinding} & \triangleq \text{Hash(LL1 NVRAM historySummary, stateHash)}
\end{align*}

The LL1 NVRAM HistorySummary Uncorrupted predicate has two conditions. First, that there exists a state hash in HashType, for which we have a witness.

(2)3. stateHash ∈ HashType

by (2)2

The second condition is that the history state binding is authenticated in the primed state.

(2)4. LL1 HistoryStateBindingAuthenticated(historyStateBinding)'

We will use the assumption that there is no change to the authentication status of any history state binding.

(3)1. \forall \text{historyStateBinding1 }\in \text{HashType} :

\begin{align*}
\text{UNCHANGED LL1 HistoryStateBindingAuthenticated(historyStateBinding1)}
\end{align*}

by (1)1

This requires that we prove the type of the history state binding.

(3)2. historyStateBinding ∈ HashType

(4)1. LL1 NVRAM historySummary ∈ HashDomain

(5)1. LL1 NVRAM historySummary ∈ HashType

(6)1. LL1 TypeInvariant

by (1)1

(6)2. QED

by (6)1, LL1 SubtypeImplication Lemma DEF LL1 Subtype Implication

(5)2. QED

by (5)1 DEF HashDomain

(4)2. stateHash ∈ HashDomain

by (2)3 DEF HashDomain

(4)3. QED

by (4)1, (4)2, HashType Safe DEF historyStateBinding

We know that the history state binding is authenticated in the unprimed state.

(3)3. LL1 HistoryStateBindingAuthenticated(historyStateBinding)

by (2)2

By applying the assumption that there is no change to the authentication status of any history state binding, we show that the history state binding is authenticated in the primed state.

(3)4. QED

by (3)1, (3)2, (3)3
Since both conditions are satisfied, the $LL1NVRAMHistorySummaryUncorrupted$ predicate is true in the primed state.

(2)5. QED  
   BY (2)3, (2)4 DEF $LL1NVRAMHistorySummaryUncorrupted$

We then consider the case in which the truth value of the predicate is false.

(1)3. CASE $LL1NVRAMHistorySummaryUncorrupted = FALSE$

To show that the value is unchanged, it suffices to show that if the value were unequal to false in the primed state, we would have a contradiction.

(2)1. SUFFICES  
   ASSUME $LL1NVRAMHistorySummaryUncorrupted' ≠ FALSE  
   PROVE FALSE  
   BY (1)3

(2)2. $LL1NVRAMHistorySummaryUncorrupted'  
   (3)1. $LL1NVRAMHistorySummaryUncorrupted' ∈ BOOLEAN  
   BY DEF $LL1NVRAMHistorySummaryUncorrupted$  
   (3)2. QED  
   BY (2)1, (3)1

We pick some state hash for which the $LL1NVRAMHistorySummaryUncorrupted$ is true in the primed state.

(2)3. PICK $stateHash ∈ HashType :  
   $LL1NVRAMHistorySummaryUncorrupted!(stateHash)'1'  
   BY (2)2 DEF $LL1NVRAMHistorySummaryUncorrupted$

We copy the definition from the let in $LL1NVRAMHistorySummaryUncorrupted$.

(2) historyStateBinding $\triangleq Hash(LL1NVRAM.historySummary, stateHash)$

The $LL1NVRAMHistorySummaryUncorrupted$ predicate has two conditions. First, that there exists a state hash in $HashType$, for which we have a witness.

(2)4. $stateHash ∈ HashType  
   BY (2)3

The second condition is that the history state binding is authenticated in the unprimed state.

(2)5. $LL1HistoryStateBindingAuthenticated(historyStateBinding)  
   We will use the assumption that there is no change to the authentication status of any history state binding.

(3)1. $∀ historyStateBinding1 ∈ HashType :  
   UNCHANGED $LL1HistoryStateBindingAuthenticated(historyStateBinding1)  
   BY (1)1

This requires that we prove the type of the history state binding.

(3)2. $historyStateBinding ∈ HashType  
   (4)1. $LL1NVRAM.historySummary ∈ HashDomain  
   (5)1. $LL1NVRAM.historySummary ∈ HashType  
      (6)1. $LL1TypeInvariant  
         BY (1)1
      (6)2. QED  
         BY (6)1, $LL1SubtypeImplicationLemma$ DEF $LL1SubtypeImplication$  
      (5)2. QED  
         BY (5)1 DEF $HashDomain$  
   (4)2. $stateHash ∈ HashDomain  
      BY (2)4 DEF $HashDomain$  
   (4)3. QED  
      BY (4)1, (4)2, $HashTypeSafe$ DEF $historyStateBinding$

We know, in our contradictory universe, that the history state binding is authenticated in the primed state.

(3)3. $LL1HistoryStateBindingAuthenticated(historyStateBinding)'  
   BY (2)3

85
By applying the assumption that there is no change to the authentication status of any history state binding, we show that the history state binding is authenticated in the unprimed state.

(3)4. QED
   BY (3)1, (3)2, (3)3
Since both conditions are satisfied, the \( LL1NVRAMHistorySummaryUncorrupted \) predicate is true in the unprimed state.

(2)6. \( LL1NVRAMHistorySummaryUncorrupted \)
   BY (2)4, (2)5 DEF \( LL1NVRAMHistorySummaryUncorrupted \)
However, we are considering the case in which the \( LL1NVRAMHistorySummaryUncorrupted \) predicate is false in the unprimed state, so we have a contradiction.

(2)7. QED
   BY (1)3, (2)6
The predicate has a boolean truth value.

(1)4. \( LL1NVRAMHistorySummaryUncorrupted \) \( \in \) BOOLEAN
   BY DEF \( LL1NVRAMHistorySummaryUncorrupted \)
(1)5. QED
   BY (1)2, (1)3, (1)4

The \( LL1RepeatOperationUnchangedObservedOutputsLemma \) states that the \( LL1RepeatOperation \) action does not change the value of \( LL1ObservedOutputs \). Together with the \( LL1RepeatOperationUnchangedAuthenticatedHistoryStateBindingsLemma \) below, this is the essence of why \( LL1RepeatOperation \) does not cause any change to the refined high-level state.

**THEOREM** \( LL1RepeatOperationUnchangedObservedOutputsLemma \) \( \triangleq \)
\[
LL1TypeInvariant \land UnforgeabilityInvariant \land InclusionInvariant \land LL1RepeatOperation \Rightarrow \text{UNCHANGED LL1ObservedOutputs}
\]
First, we assume the antecedents.

(1)1. HAVE \( LL1TypeInvariant \land UnforgeabilityInvariant \land InclusionInvariant \land LL1RepeatOperation \)
Then, we pick some input for which \( LL1RepeatOperation \) is true.

(1)2. PICK \( input \in LL1AvailableInputs \) : \( LL1RepeatOperation!(input)!1 \)
   BY (1)1 DEF \( LL1RepeatOperation \)
To simplify the writing of the proof, we re-state some of the definitions from the \( LL1RepeatOperation \) action.

(1) \( stateHash \triangleq \text{Hash}(LL1RAM.publicState, LL1RAM.privateStateEnc) \)
(1) \( historyStateBinding \triangleq \text{Hash}(LL1RAM.historySummary, stateHash) \)
(1) \( privateState \triangleq \text{SymmetricDecrypt}(LL1NVRAM.symmetricKey, LL1RAM.privateStateEnc) \)
(1) \( sResult \triangleq \text{Service}(LL1RAM.publicState, privateState, input) \)
We then assert the type safety of these definitions, with the help of the \( LL1RepeatOperationDefsTypeSafeLemma \).

(1)3. \( \land \) \( stateHash \in \text{HashType} \)
   \( \land \) \( historyStateBinding \in \text{HashType} \)
   \( \land \) \( privateState \in \text{PrivateStateType} \)
   \( \land \) \( sResult \in \text{ServiceResultType} \)
   \( \land \) \( sResult.newPublicState \in \text{PublicStateType} \)
   \( \land \) \( sResult.newPrivateState \in \text{PrivateStateType} \)
   \( \land \) \( sResult.output \in \text{OutputType} \)
(2)1. \( input \in LL1AvailableInputs \)
   BY (1)2
(2)2. \( LL1TypeInvariant \)
   BY (1)1
(2)3. QED
BY \(2\).1, \(2\).2, LL1RepeatOperationDefsTypeSafeLemma
We hide the definitions, so they don’t overwhelm the prover. We’ll pull them in as necessary below.

\(1\) HIDa DEF stateHash, historyStateBinding, privateState, sResult

From the definition of LL1RepeatOperation, we see the primed state of LL1ObservedOutputs is formed by unioning in the output from the service.

\(1\).1. LL1ObservedOutputs\' = LL1ObservedOutputs \cup \{sResult.output\}

BY \(1\).2 DEF LL1RepeatOperation, sResult, privateState

We then show that the output from the service is already in LL1ObservedOutputs.

\(1\).5. sResult.output \in LL1ObservedOutputs

Our strategy is to use the InclusionInvariant. We first have to show that all of the types are satisfied.

\(2\).1. LL1TypeInvariant

BY \(1\).1

\(2\).2. input \in InputType

\(3\).1. input \in LL1AvailableInputs

BY \(1\).2

\(3\).2. LL1AvailableInputs \subseteq InputType

BY \(2\).1 DEF LL1TypeInvariant

\(3\).3. QED

BY \(3\).1, \(3\).2

\(2\).3. \^ LL1RAM.historySummary \in HashType

\^ LL1RAM.publicState \in PublicStateType

\^ LL1RAM.privateStateEnc \in PrivateStateEncType

BY \(2\).1, LL1SubtypeImplicationLemmaDef LL1SubtypeImplication

We then have to show that the antecedents in the InclusionInvariant are satisfied.

\(2\).4. LL1NVRAM.historySummary = Hash(LL1RAM.historySummary, input)

BY \(1\).2

\(2\).5. LL1HistoryStateBindingAuthenticated(historyStateBinding)

To show that the history state binding is authenticated, we demonstrate that LL1RAM.authenticator is a sufficient witness for the existential quantifier within the definition of LL1HistoryStateBindingAuthenticated.

\(3\).1. ValidateMAC(LL1NVRAM.symmetricKey, historyStateBinding, LL1RAM.authenticator)

BY \(1\).2 DEF historyStateBinding, stateHash

\(3\).2. LL1RAM.authenticator \in LL1ObservedAuthenticators

\(4\).1. historyStateBinding \in HashType

BY \(1\).3

\(4\).2. UnforgeabilityInvariant

BY \(1\).1

\(4\).3. QED

BY \(3\).1, \(4\).1, \(4\).2 DEF UnforgeabilityInvariant

\(3\).3. QED

BY \(3\).1, \(3\).2 DEF LL1HistoryStateBindingAuthenticated

Then, we can apply the InclusionInvariant to show that the output from the service is in the set of observed outputs.

\(2\).6. QED

\(3\).1. InclusionInvariant

BY \(1\).1

\(3\).2. QED

BY \(2\).2, \(2\).3, \(2\).4, \(2\).5, \(3\).1

DEF InclusionInvariant, sResult, privateState, historyStateBinding, stateHash

Since the element being unioned into the set is already in the set, the set does not change.

\(1\).6. QED

BY \(1\).4, \(1\).5
The LL1RepeatOperationUnchangedAuthenticatedHistoryStateBindingsLemma states that the LL1RepeatOperation action does not change the set of history state bindings that have authenticators in the set LL1ObservedAuthenticators. Together with the LL1RepeatOperationUnchangedObservedOutputsLemma above, this is the essence of why LL1RepeatOperation does not cause any change to the refined high-level state.

THEOREM LL1RepeatOperationUnchangedAuthenticatedHistoryStateBindingsLemma \overset{!}{=} LL1TypeInvariant \land \text{UnforgeabilityInvariant} \land \text{InclusionInvariant} \land LL1RepeatOperation \Rightarrow
\forall \text{historyStateBinding1} \in \text{HashType} :
\text{UNCHANGED LL1HistoryStateBindingAuthenticated(historyStateBinding1)}

We assume the antecedents.

{1}1. \text{HAVE LL1TypeInvariant} \land \text{UnforgeabilityInvariant} \land \text{InclusionInvariant} \land LL1RepeatOperation

To prove the universally quantified expression, we take a new history state binding in HashType.

{1}2. \text{TAKING historyStateBinding1} \in \text{HashType}

Then, we pick some input for which LL1RepeatOperation is true.

{1}3. \text{PICK input} \in \text{LL1AvailableInputs} : LL1RepeatOperation1(input)!1

BY (1)1 \text{DEF LL1RepeatOperation}

To simplify the writing of the proof, we re-state the definitions from the LL1RepeatOperation action.

(1) \text{stateHash} \overset{!}{=} \text{Hash(LL1RAM.publicState, LL1RAM.privateStateEnc)}
(1) \text{historyStateBinding} \overset{!}{=} \text{Hash(LL1RAM.historySummary, stateHash)}
(1) \text{privateState} \overset{!}{=} \text{SymmetricDecrypt(LL1NVRAM.symmetricKey, LL1RAM.privateStateEnc)}
(1) \text{sResult} \overset{!}{=} \text{Service(LL1RAM.publicState, privateState, input)}
(1) \text{newPrivateStateEnc} \overset{!}{=} \text{SymmetricEncrypt(LL1NVRAM.symmetricKey, sResult.newPrivateState)}
(1) \text{newStateHash} \overset{!}{=} \text{Hash(sResult.newPublicState, newPrivateStateEnc)}
(1) \text{newHistoryStateBinding} \overset{!}{=} \text{Hash(LL1NVRAM.historySummary, newStateHash)}
(1) \text{newAuthenticator} \overset{!}{=} \text{GenerateMAC(LL1NVRAM.symmetricKey, newHistoryStateBinding)}

We then assert the type safety of these definitions, with the help of the LL1RepeatOperationDefsTypeSafeLemma.

{1}4. \land \text{stateHash} \in \text{HashType}
\land \text{historyStateBinding} \in \text{HashType}
\land \text{privateState} \in \text{PrivateStateType}
\land \text{sResult} \in \text{ServiceResultType}
\land \text{sResult.newPublicState} \in \text{PublicStateType}
\land \text{sResult.newPrivateState} \in \text{PrivateStateType}
\land \text{sResult.output} \in \text{OutputType}
\land \text{newPrivateStateEnc} \in \text{PrivateStateEncType}
\land \text{newStateHash} \in \text{HashType}
\land \text{newHistoryStateBinding} \in \text{HashType}
\land \text{newAuthenticator} \in \text{MACType}

(2)1. \text{input} \in \text{LL1AvailableInputs}

BY (1)3

(2)2. LL1TypeInvariant

BY (1)1

(2)3. QED

BY (2)1, (2)2, LL1RepeatOperationDefsTypeSafeLemma

We hide the definitions, so they don’t overwhelm the prover. We’ll pull them in as necessary below.

{1} \text{HIDE DEF stateHash, historyStateBinding, privateState, sResult, newPrivateStateEnc,}
\text{newStateHash, newHistoryStateBinding, newAuthenticator}

From the definition of LL1RepeatOperation, we see the primed state of LL1ObservedAuthenticators is formed by unioning in the new authenticator.
\[\text{LL1ObservedAuthenticators}' = \text{LL1ObservedAuthenticators} \cup \{\text{newAuthenticator}\}\]

\[\text{by (1)3 def LL1RepeatOperation, newAuthenticator, newHistoryStateBinding, newStateHash, newPrivateStateEnc, sResult, privateState}\]

One fact that will be useful in several places is that the symmetric key in the NVRAM has not changed.

\[\text{UNCHANGED LL1NVRAM.symmetricKey}\]

\[\text{by (1)3}\]

\[\text{QED}\]

\[\text{by (2)1}\]

To show that \(\text{LL1HistoryStateBindingAuthenticated}\) predicate is unchanged for all history state bindings, we first consider one specific history state binding, namely the the new history state binding defined in \(\text{LL1RepeatOperation}\).

\[\text{CASE historyStateBinding1 = newHistoryStateBinding}\]

First, we'll show that the new history state binding is authenticated in the unprimed state.

\[\text{LL1HistoryStateBindingAuthenticated(newHistoryStateBinding)}\]

Our strategy is to use the \text{InclusionInvariant}. We first have to show that all of the types are satisfied.

\[\text{LL1TypeInvariant}\]

\[\text{by (1)1}\]

\[\text{LL1AvailableInputs} \subseteq \text{InputType}\]

\[\text{by (3)2}\]

\[\text{qed}\]

\[\text{by (4)1, (4)2}\]

\[\text{LL1NVRAM.historySummary} = \text{Hash(LL1RAM.historySummary, input)}\]

\[\text{by (1)3}\]

\[\text{LL1HistoryStateBindingAuthenticated(historyStateBinding)}\]

To show that the history state binding is authenticated, we demonstrate that \(\text{LL1RAM.authenticator}\) is a sufficient witness for the existential quantifier within the definition of \(\text{LL1HistoryStateBindingAuthenticated}\).

\[\text{ValidateMAC(LL1NVRAM.symmetricKey, historyStateBinding, LL1RAM.authenticator)}\]

\[\text{by (1)3 def historyStateBinding, stateHash}\]

\[\text{LL1RAM.authenticator} \in \text{LL1ObservedAuthenticators}\]

\[\text{by (1)4}\]

\[\text{UnforgeabilityInvariant}\]

\[\text{by (1)1}\]

\[\text{qed}\]

\[\text{by (4)1, (5)1, (5)2 def UnforgeabilityInvariant}\]

\[\text{qed}\]

\[\text{by (4)1, (4)2 def LL1HistoryStateBindingAuthenticated}\]

Then, we can apply the \text{InclusionInvariant} to show that the new history state binding is authenticated.

\[\text{qed}\]

\[\text{by (1)1}\]
We then consider every state binding that is not equal to the new history state binding defined in $(1)$. In the unprimed state, and we'll show that they continue to be authenticated in the primed state.

We'll subdivide these into two cases. In the first case, we'll consider the history state bindings that are authenticated because the new history state binding is authenticated in both the unprimed and primed states, the new authenticator was generated as a MAC of this history state binding by $L1RepeatOperation$ in the unprimed state, and the symmetric key in the NVRAM has not changed.

Next, we'll show that the new history state binding is authenticated in the primed state.

By expanding the definition of $L1HistoryStateBindingAuthenticated$, it suffices to show that the new authenticator defined in $L1RepeatOperation$ (which we know to be in the primed set of observed authenticators) is a valid MAC for the history state binding in the primed state.

We can thus use the $MACComplete$ property to show that the generated MAC validates appropriately. To do this, we first need to prove some types.

Then, we appeal to the $MACComplete$ property in a straightforward way.

Because the new history state binding is authenticated in both the unprimed and primed states, the $L1HistoryStateBindingAuthenticated$ is unchanged for this history state binding.

We then consider every state binding that is not equal to the the new history state binding defined in $L1RepeatOperation$.

We'll subdivide these into two cases. In the first case, we'll consider the history state bindings that are authenticated in the unprimed state, and we'll show that they continue to be authenticated in the primed state.

By hypothesis, the history state binding is authenticated in the unprimed state. Thus, we can pick an authenticator in the set of observed authenticators that is a valid MAC for this history state binding.

We'll show that the new history state binding is authenticated in the primed state.

The new authenticator was generated as a MAC of this history state binding by $L1RepeatOperation$ in the unprimed state, and the symmetric key in the NVRAM has not changed.

We can thus use the $MACComplete$ property to show that the generated MAC validates appropriately. To do this, we first need to prove some types.

Then, we appeal to the $MACComplete$ property in a straightforward way.

Because the new history state binding is authenticated in both the unprimed and primed states, the $L1HistoryStateBindingAuthenticated$ is unchanged for this history state binding.

We then consider every state binding that is not equal to the the new history state binding defined in $L1RepeatOperation$.

We'll subdivide these into two cases. In the first case, we'll consider the history state bindings that are authenticated in the unprimed state, and we'll show that they continue to be authenticated in the primed state.

By hypothesis, the history state binding is authenticated in the unprimed state. Thus, we can pick an authenticator in the set of observed authenticators that is a valid MAC for this history state binding.

We'll show that the new history state binding is authenticated in the primed state.
Because the symmetric key in the NVRAM has not changed, this authenticator is also a valid MAC for this history state binding in the primed state.

(4.2) ValidateMAC(LL1NVRAM.symmetricKey', historyStateBinding1, authenticator)
    BY (1)6, (4.1)

Because the primed set of observed authenticators includes all authenticators that were in the unprimed set, this authenticator is also in the primed set of observed authenticators.

(4.3) authenticator ∈ LL1ObservedAuthenticators'

(5.1) authenticator ∈ LL1ObservedAuthenticators
    BY (4.1)

(5.2) LL1ObservedAuthenticators ⊆ LL1ObservedAuthenticators'
    BY (1)5

(5.3) QED
    BY (5.1), (5.2)

The previous two conditions are sufficient to establish that the history state binding is authenticated in the primed state.

(4.4) QED
    BY (4.2), (4.3) DEF LL1HistoryStateBindingAuthenticated

Because the history state binding is authenticated in both the unprimed and primed states, the LL1HistoryStateBindingAuthenticated is unchanged for this history state binding.

(3.2) QED
    BY (2)1, (3.1)

We’ll subdivide these into two cases. In the second case, we’ll consider the history state bindings that are unauthenticated in the unprimed state, and we’ll show that they continue to be unauthenticated in the primed state.

(2)2. CASE LL1HistoryStateBindingAuthenticated(historyStateBinding1) = FALSE

(3.1) LL1HistoryStateBindingAuthenticated(historyStateBinding1') = FALSE

To prove that the history state binding is not authenticated in the primed state, it suffices to show that none of the authenticators in the primed set of observed authenticators is a valid MAC for the history state binding.

(4.1) suffices ∀ authenticator ∈ LL1ObservedAuthenticators':
    ¬ValidateMAC(LL1NVRAM.symmetricKey', historyStateBinding1, authenticator)
    BY DEF LL1HistoryStateBindingAuthenticated

To prove the universally quantified expression, we take a new authenticator in the primed set of observed authenticators.

(4.2) TAKE authenticator ∈ LL1ObservedAuthenticators'

We’ll subdivide this into two cases. First, we consider the case in which the authenticator is in the unprimed set of authenticators. In this case, because the authenticator failed to authenticate the history state binding in the unprimed state, and the symmetric key has not changed, it immediately follows that the authenticator will not authenticate the history state binding in the primed state.

(4.3) CASE authenticator ∈ LL1ObservedAuthenticators
    BY (1)6, (2)2, (4.3) DEF LL1HistoryStateBindingAuthenticated

In the second case, we consider the new authenticator defined in LL1RepeatOperation.

(4.4) CASE authenticator = newAuthenticator

We’ll use proof by contradiction. Assume that the new authenticator is a valid MAC for the history state binding.

(5.1) suffices
    ASSUME ValidateMAC(LL1NVRAM.symmetricKey', historyStateBinding1, authenticator)
    PROVE FALSE
    OBVIOUS

By the collision resistance of MACs, it must be the case the history state binding is equal to the new history state binding defined in LL1RepeatOperation.

(5.2) historyStateBinding1 = newHistoryStateBinding
(6)1. \( \text{LL1NVRAM}.\text{symmetricKey} \in \text{SymmetricKeyType} \)
(7)1. \( \text{LL1TypeInvariant} \)
   BY (1)1
(7)2. QED
   BY (7)1, \( \text{LL1SubtypeImplicationLemma} \) DEF \( \text{LL1SubtypeImplication} \)
(6)2. \( \text{LL1NVRAM}.\text{symmetricKey}' \in \text{SymmetricKeyType} \)
   BY (1)6, (6)1
(6)3. \( \text{historyStateBinding}1 \in \text{HashType} \)
   BY (1)2
(6)4. \( \text{newHistoryStateBinding} \in \text{HashType} \)
   BY (1)4
(6)5. QED
   BY (4)4, (5)1, (6)1, (6)2, (6)3, (6)4, MACCollisionResistant DEF newAuthenticator

But we are working within a case in which the history state binding is not equal to the new history state binding defined in \( \text{LL1RepeatOperation} \). Thus, we have a contradiction.

(5)3. QED
   BY (1)8, (5)2

We’ve considered authenticators in the unprimed set of authenticators, and we’ve considered the new authenticator defined in \( \text{LL1RepeatOperation} \). Because the primed set of authenticators is the union of these two, we have exhausted the cases.

(4)5. QED
   BY (1)5, (4)3, (4)4

Because the history state binding is unauthenticated in both the unprimed and primed states, the \( \text{LL1HistoryStateBindingAuthenticated} \) is unchanged for this history state binding.

(3)2. QED
   BY (2)2, (3)1

By proving that \( \text{LL1HistoryStateBindingAuthenticated} \) is a boolean predicate, it is immediately clear that the two cases of true and false are exhaustive for this predicate.

(2)3. \( \text{LL1HistoryStateBindingAuthenticated}(\text{historyStateBinding}1) \in \text{BOOLEAN} \)
   BY DEF \( \text{LL1HistoryStateBindingAuthenticated} \)
(2)4. QED
   BY (2)1, (2)2, (2)3

Because the conclusion holds for (1) the new history state binding defined in \( \text{LL1RepeatOperation} \) and (2) every other state binding, the conclusion holds for all state bindings.

(1)9. QED
   BY (1)7, (1)8

The \( \text{LL1RAMUnforgeabilityUnchangedLemma} \) states that, if the symmetric key in the NVRAM does not change and the set of observed authenticators does not change, then the RAM’s portion of the \( \text{ExtendedUnforgeabilityInvariant} \) inductively holds when the RAM’s primed value is taken from the RAM or the disk.

**Theorem** \( \text{LL1RAMUnforgeabilityUnchangedLemma} \) \( \triangleq \)

\[
( \land \text{ExtendedUnforgeabilityInvariant} \\
\land \text{LL1TypeInvariant} \\
\land \text{LL1TypeInvariant}' \\
\land \text{LL1RAM'} \in \{ \text{LL1RAM}, \text{LL1Disk} \} \\
\land \text{LL1ObservedAuthenticators} \subseteq \text{LL1ObservedAuthenticators}' \\
\land \text{UNCHANGED} \text{ LL1NVRAM}.\text{symmetricKey} )
\Rightarrow \\
\forall \text{historyStateBinding} \in \text{HashType} : \\
\text{ValidateMAC}(\text{LL1NVRAM}.\text{symmetricKey}', \text{historyStateBinding}, \text{LL1RAM}.\text{authenticator}') \Rightarrow
\]

92
$LL1RAM'.authenticator' \in LL1ObservedAuthenticators'$

We begin by assuming the antecedent.

\begin{enumerate}
\item We have $\land$ ExtendedUnforgeabilityInvariant
\item $\land$ LL1TypeInvariant
\item $\land$ LL1TypeInvariant'
\item $\land$ LL1RAM' $\in \{LL1RAM, LL1Disk\}$
\item $\land$ LL1ObservedAuthenticators $\subseteq$ LL1ObservedAuthenticators'
\item $\land$ UNCHANGED LL1NVRAM.symmetricKey
\end{enumerate}

To prove the universally quantified expression, we take a new historyStateBinding in the HashType.

\begin{enumerate}
\item We take historyStateBinding $\in$ HashType
\item We then assume the antecedent in the nested implication.
\item $LL1RAM$'s primed state is taken from either the $LL1RAM$ or the disk.
\item $LL1RAM' \in \{LL1RAM, LL1Disk\}$
\end{enumerate}

Case 1: the $LL1RAM$'s primed state comes from the $LL1RAM$. There are three basic steps.

\begin{enumerate}
\item Case unchanged $LL1RAM$
\item First, since we take the antecedent in the nested implication and swap out unprimed variables for primed variables, since the symmetric key and authenticator have not changed.
\item Second, using the ExtendedUnforgeabilityInvariant, we show that the authenticator was in the unprimed set of observed authenticators.
\item Third, we show that the authenticator is also in the primed set of observed authenticators, since the symmetric key has not changed and set of observed authenticators includes every element in the primed state that it included in the unprimed state.
\end{enumerate}

Case 2: the RAM’s primed state comes from the disk. The proof is straightforward.

\begin{enumerate}
\item Case $LL1RAM' = LL1Disk$
\item First, since we take the antecedent in the nested implication and make two changes: (1) swap out unprimed variables for primed variables and (2) replace $LL1RAM$ with $LL1Disk$, since the primed state of the RAM comes from the unprimed state of the disk.
\end{enumerate}
Second, using the `ExtendedUnforgeabilityInvariant`, we show that the authenticator was in the unprimed set of observed authenticators.

\[(2)\] \(LL1Disk.authenticator \in LL1ObservedAuthenticators\)

\[(3)\] `ExtendedUnforgeabilityInvariant`

\[(3)\] QED

Third, we show that the authenticator is also in the primed set of observed authenticators, since the symmetric key has not changed and set of observed authenticators includes every element in the primed state that it included in the unprimed state.

\[(2)\] `LL1Disk.authenticator' = LL1Disk.authenticator`

\[(3)\] \(LL1ObservedAuthenticators \subseteq LL1ObservedAuthenticators'\)

\[(3)\] QED

The theorem is true by exhaustive case analysis.

\[(1)\] QED

By \(\langle 1\rangle 1, \langle 1\rangle 6, \langle 1\rangle 3, (3)1, (3)2\)

The `LL1DiskUnforgeabilityUnchangedLemma` states that, if the symmetric key in the NVRAM does not change and the set of observed authenticators does not change, then the disk’s portion of the `ExtendedUnforgeabilityInvariant` inductively holds when the disk’s primed value is taken from the RAM or the disk.

**Theorem** \(LL1DiskUnforgeabilityUnchangedLemma \triangleq\)

\[(\land \text{ExtendedUnforgeabilityInvariant} \land LL1TypeInvariant \land LL1TypeInvariant' \land LL1Disk' \in \{LL1RAM, LL1Disk\} \land LL1ObservedAuthenticators \subseteq LL1ObservedAuthenticators' \land \text{UNCHANGED LL1NVRAM.symmetricKey}) \Rightarrow \forall \text{historyStateBinding} \in \text{HashType}: \text{ValidateMAC}'(\text{LL1NVRAM.symmetricKey}', \text{historyStateBinding}, \text{LL1Disk.authenticator}') \Rightarrow LL1Disk.authenticator' \in LL1ObservedAuthenticators'\]

We begin by assuming the antecedent.

\[(1)\] HAVE \(\land \text{ExtendedUnforgeabilityInvariant} \land LL1TypeInvariant \land LL1TypeInvariant' \land LL1Disk' \in \{LL1RAM, LL1Disk\} \land LL1ObservedAuthenticators \subseteq LL1ObservedAuthenticators' \land \text{UNCHANGED LL1NVRAM.symmetricKey}\)

To prove the universally quantified expression, we take a new `historyStateBinding` in the `HashType`.

\[(1)\] TAKE `historyStateBinding \in HashType`

We then assume the antecedent in the nested implication.

94
Case 1: the disk’s primed state comes from the disk. There are three basic steps.

(1). First, since we take the antecedent in the nested implication and swap out unprimed variables for primed variables, since the symmetric key and authenticator have not changed.

(2). Second, using the $ExtendedUnforgeabilityInvariant$, we show that the authenticator was in the unprimed set of observed authenticators.

(3). Third, we show that the authenticator is also in the primed set of observed authenticators, since the symmetric key has not changed and set of observed authenticators includes every element in the primed state that it included in the unprimed state.

Case 2: the disk’s primed state comes from the RAM. The proof is straightforward.
Third, we show that the authenticator is also in the primed set of observed authenticators, since the symmetric key has not changed and set of observed authenticators includes every element in the primed state that it included in the unprimed state.

\[ (2) \text{QED} \]

\[ (3) 1. LL1Disk.authenticator' = LL1RAM.authenticator \]
\[ \text{BY } (1)6 \]

\[ (3) 2. LL1ObservedAuthenticators \subseteq LL1ObservedAuthenticators' \]
\[ \text{BY } (1)1 \]

\[ (3) 3. \text{QED} \]
\[ \text{BY } (2)2, (3)1, (3)2 \]

The theorem is true by exhaustive case analysis.

\[ (1) \text{QED} \]
\[ \text{BY } (1)4, (1)5, (1)6 \]

The **InclusionUnchangedLemma** states that, if there are no changes to the NVRAM, to the set of observed outputs, or to the authentication status of any history state binding, then the **InclusionInvariant** holds inductively from the unprimed state to the primed state.

**THEOREM**  **InclusionUnchangedLemma**  
\[ \Delta \]
\[ \begin{align*}
& (\land \text{InclusionInvariant} \\
& \land LL1TypeInvariant \\
& \land LL1TypeInvariant' \\
& \land \text{UNCHANGED } (LL1NVRAM, LL1ObservedOutputs) \\
& \land \forall \text{historyStateBinding1 }\in \text{HashType} : \\
& \quad \text{UNCHANGED } LL1\text{HistoryStateBindingAuthenticated(historyStateBinding1)}
\end{align*} \]
\[ \Rightarrow \text{InclusionInvariant}' \]

We begin by assuming the antecedent.

\[ (1) 1. \text{HAVE} \quad (\land \text{InclusionInvariant} \\
& \land LL1TypeInvariant \\
& \land LL1TypeInvariant' \\
& \land \text{UNCHANGED } LL1NVRAM, LL1ObservedOutputs \\
& \land \forall \text{historyStateBinding1 }\in \text{HashType} : \\
& \quad \text{UNCHANGED } LL1\text{HistoryStateBindingAuthenticated(historyStateBinding1)} \]

To prove the universally quantified expression, we take a new set of variables in the appropriate types. For the **take** step to be meaningful to the prover, first we have to tell the prover to expand the definition of **InclusionInvariant**, so it will see the universally quantified expression therein.

\[ (1) \text{USE DEF InclusionInvariant} \]
\[ (1) 2. \text{TAK} \\
& \text{input }\in \text{InputType}, \\
& \text{historySummary }\in \text{HashType}, \\
& \text{publicState }\in \text{PublicStateType}, \\
& \text{privateStateEnc }\in \text{PrivateStateEncType} \]

To simplify the writing of the proof, we re-state the definitions from the **InclusionInvariant**.

\[ (1) \text{stateHash }\Delta \text{Hash(publicState, privateStateEnc)} \]
\[ (1) \text{historyStateBinding }\Delta \text{Hash(historySummary, stateHash)} \]
\[ (1) \text{privateState }\Delta \text{SymmetricDecrypt(LL1NVRAM.symmetricKey, privateStateEnc)} \]
\[ (1) \text{sResult }\Delta \text{Service(publicState, privateState, input)} \]
\[ (1) \text{newPrivateStateEnc }\Delta \]
\[ \text{SymmetricEncrypt(LL1NVRAM.symmetricKey, sResult.newPrivateState)} \]
\[ (1) \text{newStateHash }\Delta \text{Hash(sResult.newPublicState, newPrivateStateEnc)} \]
newHistoryStateBinding \triangleq Hash(LL1NVRAM.historySummary, newStateHash)

We then assert the type safety of these definitions, with the help of the InclusionInvariantDefsTypeSafeLemma.

\[ \begin{align*}
\land & \quad \text{stateHash} \in \text{HashType} \\
\land & \quad \text{historyStateBinding} \in \text{HashType} \\
\land & \quad \text{privateState} \in \text{PrivateStateType} \\
\land & \quad \text{sResult} \in \text{ServiceResultType} \\
\land & \quad \text{sResult}.\text{newPublicState} \in \text{PublicStateType} \\
\land & \quad \text{sResult}.\text{newPrivateState} \in \text{PrivateStateType} \\
\land & \quad \text{sResult}.\text{output} \in \text{OutputType} \\
\land & \quad \text{newPrivateStateEnc} \in \text{PrivateStateEncType} \\
\land & \quad \text{newStateHash} \in \text{HashType} \\
\land & \quad \text{newHistoryStateBinding} \in \text{HashType}
\end{align*} \]

We hide the definitions, so they don’t overwhelm the prover. We’ll pull them in as necessary below.

\[ \begin{align*}
\land & \quad \text{LL1NVRAM}.\text{historySummary}' = Hash(\text{historySummary}, \text{input}) \\
\land & \quad \text{LL1HistoryStateBindingAuthenticated}(\text{historyStateBinding})'
\end{align*} \]

We prove the two conjuncts in the antecedent of the InclusionInvariant. Each follows as a straightforward consequence of the fact that LL1NVRAM and LL1ObservedAuthenticators have not changed.

\[ \begin{align*}
\land & \quad \text{result}.\text{output}' \in \text{LL1ObservedOutputs}'
\end{align*} \]

Step 1: Before the action, the output of the service is in the set of observed outputs. This follows because the InclusionInvariant is true in the unprimed state.

\[ \begin{align*}
\land & \quad \text{LL1NVRAM}.\text{historySummary} = Hash(\text{historySummary}, \text{input}) \\
\land & \quad \text{LL1ObservedOutputs} = Hash(\text{historySummary}, \text{input})
\end{align*} \]

(4)1. LL1NVRAM.historySummary' = Hash(historySummary, input)

BY (1)4

(4)2. unchanged LL1NVRAM.historySummary

(5)1. unchanged LL1NVRAM

BY (1)1

(5)2. QED

BY (5)1

(4)3. QED

BY (4)1, (4)2

(3)2. LL1HistoryStateBindingAuthenticated(historyStateBinding)
The history state binding is authenticated in the primed state, by hypothesis.

\((4)\) 1. \(\text{LL1HistoryStateBindingAuthenticated}(\text{historyStateBinding})'\)
   By \((1)4\)

The authentication status of the history state binding has not changed, because this status has not changed for any history state binding in \(\text{HashType}\).

\((4)\) 2. UNCHANGED \(\text{LL1HistoryStateBindingAuthenticated}(\text{historyStateBinding})\)
   We have to show that the history state binding has the appropriate type.

\((5)\) 1. \(\text{historyStateBinding} \in \text{HashType}\)
   By \((1)3\)

The authentication status of all input state bindings has not changed, as assumed by the lemma.

\((5)\) 2. \(\forall \text{historyStateBinding1} \in \text{HashType} : \)
   UNCHANGED \(\text{LL1HistoryStateBindingAuthenticated}(\text{historyStateBinding1})\)
   By \((1)1\)

The conclusion follows directly.

\((5)\) 3. QED
   By \((5)1, (5)2\)

Since the history state binding is authenticated in the primed state, and since its authentication status has not changed, it is also authenticated in the unprimed state.

\((4)\) 3. QED
   By \((4)1, (4)2\)

We can then use the \(\text{InclusionInvariant}\) to prove that the output of the service is in the set of observed outputs in the unprimed state.

\((3)\) 3. \(\text{InclusionInvariant}\)
   By \((1)1\)

\((3)\) 4. QED
   By \((1)2, (3)1, (3)2, (3)3\)
   DEF \(\text{InclusionInvariant}, \text{sResult, privateState, historyStateBinding, stateHash}\)

Step 2: The output of the service does not change.

\((2)\) 2. UNCHANGED \(\text{sResult.output}\)
   \((3)\) 1. UNCHANGED \(\text{privateState}\)
      \((4)\) 1. UNCHANGED \(\text{LL1NVRAM.symmetricKey}\)
         \((5)\) 1. UNCHANGED \(\text{LL1NVRAM}\)
            By \((1)1\)
         \((5)\) 2. QED
            By \((5)1\)
      \((4)\) 2. QED
         By \((4)1\) DEF \(\text{privateState}\)
   \((3)\) 2. UNCHANGED \(\text{sResult}\)
      By \((3)1\) DEF \(\text{sResult}\)
   \((3)\) 3. QED
      By \((3)2\)

Step 3: The set of observed outputs does not change.

\((2)\) 3. UNCHANGED \(\text{LL1ObservedOutputs}\)
   By \((1)1\)
\((2)\) 4. QED
   By \((2)1, (2)2, (2)3\)

Following is the proof of the second conjunct.
To prove that the new history state binding is authenticated in the primed state, we prove that the new history state binding does not change, and we prove that the new history state binding was authenticated in the unprimed state. Since, by assumption of the lemma, the authentication status of any history state binding does not change, the new history state binding is authenticated in the primed state.

(1)6. $\text{LL1HistoryStateBindingAuthenticated}(\text{newHistoryStateBinding})'$

One fact we’ll need several times is that the symmetric key in the NVRAM has not changed, so we’ll prove this once up front.

(2)1. UNCHANGED $\text{LL1NVRAM.symmetricKey}$
   (3)1. UNCHANGED $\text{LL1NVRAM}$
      BY (1)1
   (3)2. QED
      BY (3)1

In the unprimed state, the new history state binding was authenticated. This follows because the InclusionInvariant is true in the unprimed state.

(2)2. $\text{LL1HistoryStateBindingAuthenticated}(\text{newHistoryStateBinding})$

We prove the two conjuncts in the antecedent of the InclusionInvariant. The first conjunct follows as a straightforward consequence of the fact that NVRAM has not changed.

(3)1. $\text{LL1NVRAM.historySummary} = \text{Hash(historySummary, input)}$
   (4)1. $\text{LL1NVRAM.historySummary}' = \text{Hash(historySummary, input)}$
      BY (1)4
   (4)2. UNCHANGED $\text{LL1NVRAM.historySummary}$
      (5)1. UNCHANGED $\text{LL1NVRAM}$
         BY (1)1
      (5)2. QED
         BY (5)1
   (4)3. QED
      BY (4)1, (4)2

Proving the second conjunct is more involved. We’ll show that the history state binding is authenticated in the primed state and that its authentication status has not changed.

(3)2. $\text{LL1HistoryStateBindingAuthenticated}(\text{historyStateBinding})$

The history state binding is authenticated in the primed state, by hypothesis.

(4)1. $\text{LL1HistoryStateBindingAuthenticated}(\text{historyStateBinding})'$
   BY (1)4

The authentication status of the history state binding has not changed, because this status has not changed for any history state binding in HashType.

(4)2. UNCHANGED $\text{LL1HistoryStateBindingAuthenticated}(\text{historyStateBinding})$

We have to show that the history state binding has the appropriate type.

(5)1. historyStateBinding $\in$ HashType
   BY (1)3

The authentication status of all input state bindings has not changed, as assumed by the lemma.

(5)2. $\forall$ historyStateBinding1 $\in$ HashType :
   UNCHANGED $\text{LL1HistoryStateBindingAuthenticated}(\text{historyStateBinding1})$
   BY (1)1

The conclusion follows directly.

(5)3. QED
   BY (5)1, (5)2

Since the history state binding is authenticated in the primed state, and since its authentication status has not changed, it is also authenticated in the unprimed state.

(4)3. QED
   BY (4)1, (4)2

99
We can then use the InclusionInvariant to prove that the new history state binding was authenticated in the unprimed state.

(3)3. InclusionInvariant
   BY (1)1
(3)4. QED
   BY (1)2, (3)1, (3)2, (3)3
   DEF InclusionInvariant, newHistoryStateBinding, newStateHash, newPrivateStateEnc,
   sResult, privateState, historyStateBinding, stateHash
(2)3. UNCHANGED LL1HistoryStateBindingAuthenticated(newHistoryStateBinding)
The new history state binding has not changed.

(3)1. UNCHANGED newHistoryStateBinding
    (4)1. UNCHANGED LL1NVRAM.historySummary
       (5)1. UNCHANGED LL1NVRAM
          BY (1)1
       (5)2. QED
          BY (5)1
    (4)2. UNCHANGED newStateHash
       (5)1. UNCHANGED sResult
          (6)1. UNCHANGED privateState
             BY (2)1 DEF privateState
          (6)2. QED
             BY (6)1 DEF sResult
       (5)2. UNCHANGED sResult.newPublicState
          BY (5)1
    (5)3. UNCHANGED newPrivateStateEnc
       (6)1. UNCHANGED sResult.newPrivateState
          BY (5)1
       (6)2. QED
          BY (2)1, (6)1 DEF newPrivateStateEnc
       (5)4. QED
          BY (5)2, (5)3 DEF newStateHash
    (4)3. QED
       BY (4)1, (4)2 DEF newHistoryStateBinding
The new history state binding has the appropriate type.

(3)2. newHistoryStateBinding ∈ HashType
   BY (1)3
The authentication status of all input state bindings has not changed, as assumed by the lemma.

(3)3. ∀ historyStateBinding1 ∈ HashType :
       UNCHANGED LL1HistoryStateBindingAuthenticated(historyStateBinding1)
   BY (1)1
(3)4. QED
      BY (3)1, (3)2, (3)3
(2)4. QED
      BY (2)2, (2)3
Each of the conjuncts is true, so the conjunction is true.

(1)7. QED
   BY (1)5, (1)6

The CardinalityUnchangedLemma states that, if there are no changes to the NVRAM or to the authentication status of any history state binding, then the CardinalityInvariant holds inductively from the unprimed state to the primed state.
THEOREM \textit{CardinalityUnchangedLemma} \quad \Delta \\
\begin{align*}
\land LL1\text{TypeInvariant} & \\
\land \text{CardinalityInvariant} & \\
\land \text{UNCHANGED LL1NVRAM} & \\
\land \forall \text{historyStateBinding} \in \text{HashType} : & \\
\text{UNCHANGED LL1HistoryStateBindingAuthenticated}\left(\text{historyStateBinding}\right) & \\
\Rightarrow & \\
\text{CardinalityInvariant}' & \\
\end{align*}

We begin by assuming the antecedent.

\begin{enumerate}
\item \text{H}AV\ E \land LL1\text{TypeInvariant} \\
\land \text{CardinalityInvariant} \\
\land \text{UNCHANGED LL1NVRAM} \\
\land \forall \text{historyStateBinding} \in \text{HashType} : \\
\text{UNCHANGED LL1HistoryStateBindingAuthenticated}\left(\text{historyStateBinding}\right)
\end{enumerate}

To prove the universally quantified expression, we take a new set of variables in the appropriate types. For the \textsc{take} step to be meaningful to the prover, first we have to tell the prover to expand the definition of \textit{CardinalityInvariant}, so it will see the universally quantified expression therein.

\begin{enumerate}
\item \textsc{use} def \textit{CardinalityInvariant} \\
\item \textsc{take} \text{historySummary} \in \text{HashType}, \text{stateHash} \in \text{HashType}
\end{enumerate}

To simplify the writing of the proof, we re-state the definition from the \textit{CardinalityInvariant}.

\begin{enumerate}
\item \text{historyStateBinding} \Delta \begin{align*}
\text{Hash}\left(\text{historySummary}, \text{stateHash}\right)
\end{align*}
\end{enumerate}

We then assert the type safety of these definitions, with the help of the \textit{CardinalityInvariantDefsTypeSafeLemma}.

\begin{enumerate}
\item \text{historyStateBinding} \in \text{HashType} \\
\text{by} \begin{align*}
\item \text{2}
\end{align*}
\end{enumerate}

The \textit{CardinalityInvariant} states an implication. To prove this, it suffices to assume the antecedent and prove the consequent.

\begin{enumerate}
\item \textsc{s}uffi\textsc{c}es \text{assume} \land LL1\text{NVRAMHistorySummaryUncorrupted}' \\
\land LL1\text{HistoryStateBindingAuthenticated}\left(\text{historyStateBinding}\right)
\end{enumerate}

\textsc{prove} \quad \text{HashCardinality}\left(\text{historySummary}\right) \leq \text{HashCardinality}\left(LL1\text{NVRAM}.\text{historySummary}'\right)

\textsc{obvious}

We hide the definitions, so they don’t overwhelm the prover. We’ll pull them in as necessary below.

\begin{enumerate}
\item \textsc{h}ide def \textit{CardinalityInvariant} \\
\item \textsc{h}ide def \textit{historyStateBinding}
\end{enumerate}

To prove the inequality in the primed state, we prove two things: (1) Before the action, the inequality held. (2) The history summary in the NVRAM does not change.

The inequality held before the action because the \textit{CardinalityInvariant} was true in the unprimed state.

\begin{enumerate}
\item \textsc{h}ide def \textit{LL1NVRAMHistorySummaryUncorrupted} \\
\item \textsc{h}ide def \textit{LL1NVRAMHistorySummaryUncorrupted'}
\end{enumerate}

\text{We'll prove the two antecedents of the \textit{CardinalityInvariant} in the unprimed state. First, we'll prove that the LL1NVRAMHistorySummaryUncorrupted predicate is true.}

\begin{enumerate}
\item \textsc{L}L1\text{NVRAMHistorySummaryUncorrupted} \\
\text{The LL1NVRAMHistorySummaryUncorrupted predicate is true in the primed state, by hypothesis.}
\item \textsc{L}L1\text{NVRAMHistorySummaryUncorrupted'} \\
\text{by} \begin{align*}
\item \text{1}4
\end{align*}
\end{enumerate}

The LL1NVRAMHistorySummaryUncorrupted predicate has not changed, thanks to the LL1NVRAMHistorySummaryUncorruptedUnchangedLemma.

\begin{enumerate}
\item \textsc{u}nch\textsc{anged} LL1\text{NVRAMHistorySummaryUncorrupted} \\
\text{by} \begin{align*}
\item \text{1}1, LL1\text{NVRAMHistorySummaryUncorruptedUnchangedLemma}
\end{align*}
\end{enumerate}
Since the $LL_1$NVRAMHistorySummaryUncorrupted predicate is true in the primed state, and since the $LL_1$NVRAMHistorySummaryUncorrupted predicate has not changed, the predicate is also true in the unprimed state.

Then, we'll prove that the history state binding is authenticated in the unprimed state. This is the second antecedent of the CardinalityInvariant.

We have to show that the history state binding has the appropriate type.

The conclusion follows directly.

We can then use the CardinalityInvariant to prove that the inequality held in the unprimed state.

Step 2: The history summary in the NVRAM does not change.

The UniquenessUnchangedLemma states that, if there are no changes to the NVRAM or to the authentication status of any history state binding, then the UniquenessInvariant holds inductively from the unprimed state to the primed state.

**THEOREM** UniquenessUnchangedLemma \( \triangleq \)

\[ ( \land LL_1TypeInvariant ) \land UniquenessInvariant \]
∧ UNCHANGED LL1NVRAM
∧ ∀ historyStateBinding ∈ HashType :
    UNCHANGED LL1HistoryStateBindingAuthenticated(historyStateBinding)
⇒ UniquenessInvariant’

We begin by assuming the antecedent.

1. HAVE ∧ LL1TypeInvariant
∧ UniquenessInvariant
∧ UNCHANGED LL1NVRAM
∧ ∀ historyStateBinding ∈ HashType :
    UNCHANGED LL1HistoryStateBindingAuthenticated(historyStateBinding)

To prove the universally quantified expression, we take a new set of variables in the appropriate types. For the take step to be meaningful to the prover, first we have to tell the prover to expand the definition of UniquenessInvariant, so it will see the universally quantified expression therein.

1. USE DEF UniquenessInvariant
1.2. TAKE stateHash1, stateHash2 ∈ HashType

To simplify the writing of the proof, we re-state the definitions from the UniquenessInvariant.

1. historyStateBinding1 = Hash(LL1NVRAM.historySummary, stateHash1)
1. historyStateBinding2 = Hash(LL1NVRAM.historySummary, stateHash2)

We then assert the type safety of these definitions, with the help of the UniquenessInvariantDefsTypeSafeLemma.

1.3. ∧ historyStateBinding1 ∈ HashType
∧ historyStateBinding2 ∈ HashType
1.4. LL1TypeInvariant
BY (1)1
2. QED
BY (1)2, (2)1, UniquenessInvariantDefsTypeSafeLemma

The UniquenessInvariant states an implication. To prove this, it suffices to assume the antecedent and prove the consequent.

1.4. SUFFICES
ASSUME ∧ LL1HistoryStateBindingAuthenticated(historyStateBinding1)’
∧ LL1HistoryStateBindingAuthenticated(historyStateBinding2)’
PROVE stateHash1 = stateHash2

OBVIOUS

We hide the definitions, so they don’t overwhelm the prover. We’ll pull them in as necessary below.

1. HIDE DEF UniquenessInvariant
1. HIDE DEF historyStateBinding1, historyStateBinding2

First we’ll show that history state binding 1 is authenticated in the unprimed state.

1.5. LL1HistoryStateBindingAuthenticated(historyStateBinding1)

History state binding 1 is authenticated in the primed state, by hypothesis.

1.5. LL1HistoryStateBindingAuthenticated(historyStateBinding1)’
BY (1)4

The authentication status of history state binding 1 has not changed, because this status has not changed for any history state binding in HashType.

2.2. UNCHANGED LL1HistoryStateBindingAuthenticated(historyStateBinding1)

We have to show that the history state binding has not changed, which is not obvious because it is defined in terms of the history summary in the NVRAM, so we have to derive this from the fact that the NVRAM has not changed.

3.1. UNCHANGED historyStateBinding1
4.1. UNCHANGED LL1NVRAM.historySummary
5.1. UNCHANGED LL1NVRAM
BY (1)1
Then, we have to show that history state binding 1 has the appropriate type.

(3.2) \textit{historyStateBinding}1 \in \textit{HashType}

\text{by } \langle 1 \rangle

The authentication status of all input state bindings has not changed, as assumed by the lemma.

(3.3) \forall \textit{historyStateBinding} \in \textit{HashType} :

\text{UNCHANGED } LL1\text{HistoryStateBindingAuthenticated}(\textit{historyStateBinding})

\text{by } \langle 1 \rangle

The conclusion follows directly.

(3.4) QED

\text{by } \langle 3 \rangle 1, \langle 3 \rangle 2, \langle 3 \rangle 3

Since history state binding 1 is authenticated in the primed state, and since its authentication status has not changed, it is also authenticated in the unprimed state.

(2.3) QED

\text{by } \langle 2 \rangle 1, \langle 2 \rangle 2

The same argument holds for history state binding 2.

(1.6) \textit{LL1HistoryStateBindingAuthenticated}(\textit{historyStateBinding}2)

History state binding 2 is authenticated in the primed state, by hypothesis.

(2.1) \textit{LL1HistoryStateBindingAuthenticated}(\textit{historyStateBinding}2)'

\text{by } \langle 1 \rangle 4

The authentication status of history state binding 2 has not changed, because this status has not changed for any history state binding in \textit{HashType}.

(2.2) \text{UNCHANGED } LL1\text{HistoryStateBindingAuthenticated}(\textit{historyStateBinding}2)

We have to show that the history state binding has not changed, which is not obvious because it is defined in terms of the history summary in the \textit{NVRAM}, so we have to derive this from the fact that the \textit{NVRAM} has not changed.

(3.1) \text{UNCHANGED } \textit{historyStateBinding}2

(4.1) \text{UNCHANGED } LL1\text{NVRAM}.\text{historySummary}

(5.1) \text{UNCHANGED } LL1\text{NVRAM}

\text{by } \langle 1 \rangle 1

(5.2) QED

\text{by } \langle 5 \rangle 1

(4.2) QED

\text{by } \langle 4 \rangle 1 \text{ DEF } \textit{historyStateBinding}2

Then, we have to show that history state binding 2 has the appropriate type.

(3.2) \textit{historyStateBinding}2 \in \textit{HashType}

\text{by } \langle 1 \rangle

The authentication status of all input state bindings has not changed, as assumed by the lemma.

(3.3) \forall \textit{historyStateBinding} \in \textit{HashType} :

\text{UNCHANGED } LL1\text{HistoryStateBindingAuthenticated}(\textit{historyStateBinding})

\text{by } \langle 1 \rangle

The conclusion follows directly.

(3.4) QED

\text{by } \langle 3 \rangle 1, \langle 3 \rangle 2, \langle 3 \rangle 3

Since history state binding 1 is authenticated in the primed state, and since its authentication status has not changed, it is also authenticated in the unprimed state.

(2.3) QED
By (2)1, (2)2
Since the UniquenessInvariant holds in the unprimed state, it follows directly that the two state hashes are equal.

(1)7. QED
(2)1. stateHash1 ∈ HashType
   By (1)2
(2)2. stateHash2 ∈ HashType
   By (1)2
(2)3. UniquenessInvariant
   By (1)1
(2)4. QED
   By (1)5, (1)6, (2)1, (2)2, (2)3
DEF UniquenessInvariant, historyStateBinding1, historyStateBinding2

The UnchangedAuthenticatedHistoryStateBindingsLemma states that if the NVRAM and the set of observed authenticators does not change, then there is no change to the set of history state bindings that have authenticators in the set LL1ObservedAuthenticators.

THEOREM UnchangedAuthenticatedHistoryStateBindingsLemma ≜
UNCHANGED (LL1NVRAM, LL1ObservedAuthenticators) ⇒
∀ historyStateBinding ∈ HashType :
UNCHANGED LL1HistoryStateBindingAuthenticated(historyStateBinding)

We assume the antecedents.

(1)1. HAVE UNCHANGED (LL1NVRAM, LL1ObservedAuthenticators)
To prove the universally quantified expression, we take a new history state binding in HashType.

(1)2. TAKE historyStateBinding ∈ HashType
One fact that will be useful in several places is that the symmetric key in the NVRAM has not changed.

(1)3. UNCHANGED LL1NVRAM.symmetricKey
   (2)1. UNCHANGED LL1NVRAM
      By (1)1
   (2)2. QED
      By (2)1
We’ll subdivide these into two cases. In the first case, we’ll consider the history state bindings that are authenticated in the unprimed state, and we’ll show that they continue to be authenticated in the primed state.

(1)4. CASE LL1HistoryStateBindingAuthenticated(historyStateBinding) = TRUE
   (2)1. LL1HistoryStateBindingAuthenticated(historyStateBinding)′ = TRUE
      By hypothesis, the history state binding is authenticated in the unprimed state. Thus, we can pick an authenticator in the set of observed authenticators that is a valid MAC for this history state binding.
      (3)1. PICK authenticator ∈ LL1ObservedAuthenticators :
         ValidateMAC(LL1NVRAM.symmetricKey, historyStateBinding, authenticator)
         By (1)4 DEF LL1HistoryStateBindingAuthenticated
      Because the symmetric key in the NVRAM has not changed, this authenticator is also a valid MAC for this history state binding in the primed state.
      (3)2. ValidateMAC(LL1NVRAM.symmetricKey′, historyStateBinding, authenticator)
         By (1)3, (3)1
      Because the set of observed authenticators has not changed, this authenticator is also in the primed set of observed authenticators.
      (3)3. authenticator ∈ LL1ObservedAuthenticators′
      (4)1. authenticator ∈ LL1ObservedAuthenticators
         By (3)1
The previous two conditions are sufficient to establish that the history state binding is authenticated in the primed state.

By (3)2, (3)3 DEF \( LL_1 \text{HistoryStateBindingAuthenticated} \)

Because the history state binding is authenticated in both the unprimed and primed states, the \( LL_1 \text{HistoryStateBindingAuthenticated} \) is unchanged for this history state binding.

(2)2. QED
BY (1)4, (2)1

In the second case, we’ll consider the history state bindings that are unauthenticated in the unprimed state, and we’ll show that they continue to be unauthenticated in the primed state.

(1)5. CASE \( LL_1 \text{HistoryStateBindingAuthenticated}(\text{historyStateBinding}) = \text{FALSE} \)
(2)1. \( LL_1 \text{HistoryStateBindingAuthenticated}(\text{historyStateBinding})' = \text{FALSE} \)

To prove that the history state binding is not authenticated in the primed state, it suffices to show that none of the state authenticators in the primed set of observed authenticators is a valid MAC for the history state binding.

(3)1. SUFFICES \( \forall \text{authenticator} \in LL_1 \text{ObservedAuthenticators}' : \)
\(\neg \text{ValidateMAC}(LL_1 \text{NVRAM}.\text{symmetricKey}', \text{historyStateBinding}, \text{authenticator}) \)
BY DEF \( LL_1 \text{HistoryStateBindingAuthenticated} \)

To prove the universally quantified expression, we take a new authenticator in the primed set of observed authenticators.

(3)2. TAKE \( \text{authenticator} \in LL_1 \text{ObservedAuthenticators}' \)

Because the set of observed authenticators has not changed, this authenticator is also in the unprimed set of observed authenticators.

(3)3. \( \text{authenticator} \in LL_1 \text{ObservedAuthenticators} \)

(4)1. UNCHANGED \( LL_1 \text{ObservedAuthenticators} \)
BY (1)1
(4)2. QED
BY (3)2, (4)1

Because the authenticator failed to authenticate the history state binding in the unprimed state, and the symmetric key has not changed, it immediately follows that the authenticator will not authenticate the history state binding in the primed state.

(3)4. QED
BY (1)3, (1)5, (3)3 DEF \( LL_1 \text{HistoryStateBindingAuthenticated} \)

Because the history state binding is unauthenticated in both the unprimed and primed states, the \( LL_1 \text{HistoryStateBindingAuthenticated} \) is unchanged for this history state binding.

(2)2. QED
BY (1)5, (2)1

By proving that \( LL_1 \text{HistoryStateBindingAuthenticated} \) is a boolean predicate, it is immediately clear that the two cases of true and false are exhaustive for this predicate.

(1)6. \( LL_1 \text{HistoryStateBindingAuthenticated}(\text{historyStateBinding}) \in \text{BOOLEAN} \)
BY DEF \( LL_1 \text{HistoryStateBindingAuthenticated} \)

(1)7. QED
BY (1)4, (1)5, (1)6
4.5 Proof of Unforgeability Invariance in Memoir-Basic

This module proves that the UnforgeabilityInvariant is an inductive invariant of the Memoir-Basic spec.

EXTENDS MemoirLL1InvariantLemmas

Because the spec allows the data on the disk to be read into the RAM, proving UnforgeabilityInvariant of the RAM requires also proving an analogous property for the disk. Thus, we first prove the that the ExtendedUnforgeabilityInvariant is an invariant of the spec.

THEOREM ExtendedUnforgeabilityInvariant \(\triangleq LL1Spec \Rightarrow \Box ExtendedUnforgeabilityInvariant\)

This proof will require the LL1TypeInvariant. Fortunately, the LL1TypeSafe theorem has already proven that the Memoir-Basic spec satisfies its type invariant.

(1)1. LL1Spec \(\Rightarrow \Box LL1TypeInvariant\)
   BY LL1TypeSafe

The top level of the proof is boilerplate TLA+ for an Inval-style proof. First, we prove that the initial state satisfies ExtendedUnforgeabilityInvariant. Second, we prove that the LL1Next predicate inductively preserves ExtendedUnforgeabilityInvariant. Third, we use temporal induction to prove that these two conditions satisfy the ExtendedUnforgeabilityInvariant over all behaviors.

(1)2. LL1Init \(\wedge LL1TypeInvariant \Rightarrow ExtendedUnforgeabilityInvariant\)

First, we assume the antecedents.

(2)1. HAVE LL1Init \(\wedge LL1TypeInvariant\)

Then, we pick some symmetric key for which LL1Init is true.

(2)2. PICK symmetricKey \(\in SymmetricKeyType : LL1Init!\langle symmetricKey \rangle!1\)
   BY (2)1 DEF LL1Init

To simplify the writing of the proof, we re-state some of the definitions from LL1Init. We don’t need all of them for this proof, so we only re-state the ones we need.

(2) initialPrivateStateEnc \(\triangleq SymmetricEncrypt\langle symmetricKey, InitialPrivateState \rangle\)
(2) initialStateHash \(\triangleq Hash\langle InitialPublicState, initialPrivateStateEnc \rangle\)
(2) initialHistoryStateBinding \(\triangleq Hash\langle BaseHashValue, initialStateHash \rangle\)
(2) initialAuthenticator \(\triangleq GenerateMAC\langle symmetricKey, initialHistoryStateBinding \rangle\)

We hide the definitions, so they don’t overwhelm the prover. We’ll pull them in as necessary below.

(2) HIDE DEF initialPrivateStateEnc, initialStateHash, initialHistoryStateBinding, initialAuthenticator

To prove the universally quantified expression, we take a new hash. For the take step to be meaningful to the prover, first we have to tell the prover to expand the definition of ExtendedUnforgeabilityInvariant, so it will see the universally quantified expression therein.

(2) USE DEF ExtendedUnforgeabilityInvariant
(2)3. TAKE historyStateBinding \(\in HashType\)

We will prove each of the conjuncts separately. Following is the proof of the unforgeability invariant for the RAM. It follows directly from the definition of LL1Init.

(2)4. ValidateMAC_PP(LL1NVRAM, symmetricKey, historyStateBinding, LL1RAM, authenticator) \(\Rightarrow\)
   LL1RAM, authenticator \(\in LL1ObservedAuthenticators\)
   (3)1. HAVE ValidateMAC_PP(LL1NVRAM, symmetricKey, historyStateBinding, LL1RAM, authenticator)
(3)2. LL1RAM, authenticator = initialAuthenticator
   BY (2)2 DEF initialAuthenticator, initialHistoryStateBinding, initialStateHash, initialPrivateStateEnc
(3)3. LL1ObservedAuthenticators = \(\{\text{initialAuthenticator}\}\)
   BY (2)2 DEF initialAuthenticator, initialHistoryStateBinding, initialStateHash, initialPrivateStateEnc
(3)4. QED
For the induction step, we will need the type invariant to be true in both the unprimed and primed states.

Following is the proof of the unforgeability invariant for the disk. It follows directly from the definition of $LL1Init$.

(2) 5. $\text{ValidateMAC}(\text{LL1NVRAM}.\text{symmetricKey}', \text{historyStateBinding}, \text{LL1Disk}.\text{authenticator}') \Rightarrow$

\begin{align*}
\text{LL1Disk}.\text{authenticator}' & \in \text{LL1ObservedAuthenticators}' \\
\text{LL1NVRAM}.\text{symmetricKey}' & \in \text{LL1ObservedAuthenticators}'
\end{align*}

(3) 1. $\text{HAVE ValidateMAC}(\text{LL1NVRAM}.\text{symmetricKey}, \text{historyStateBinding}, \text{LL1Disk}.\text{authenticator})$

(3) 2. $\text{LL1Disk}.\text{authenticator} = \text{initialAuthenticator}$

By (2) 2 DEF initialAuthenticator, initialHistoryStateBinding, initialStateHash, initialPrivateStateEnc

(3) 3. $\text{LL1ObservedAuthenticators} = \{\text{initialAuthenticator}\}$

By (2) 2 DEF initialAuthenticator, initialHistoryStateBinding, initialStateHash, initialPrivateStateEnc

(3) 4. QED

By (3) 2, (3) 3

(2) 6. QED

By (2) 4, (2) 5

For the induction step, we will need the type invariant to be true in both the unprimed and primed states.

(1) 3. $(\wedge \text{ExtendedUnforgeabilityInvariant}'$

$\wedge [\text{LL1Next}]_{\text{LL1Vars}}$

$\wedge \text{LL1TypeInvariant}$

$\wedge \text{LL1TypeInvariant}'$)

$\Rightarrow$

$\text{ExtendedUnforgeabilityInvariant}'$

First, we assume the antecedents.

(2) 1. $\text{HAVE ExtendedUnforgeabilityInvariant} \land [\text{LL1Next}]_{\text{LL1Vars}} \land \text{LL1TypeInvariant} \land \text{LL1TypeInvariant}'$

The induction step includes two cases: stuttering and LL1Next actions. The stuttering case is a straightforward application of the $LL1RAMUnforgeabilityUnchangedLemma$ and the $LL1DiskUnforgeabilityUnchangedLemma$.

(2) 2. CASEUNCHANGED LL1Vars

(3) 1. $\forall \text{historyStateBinding} \in \text{HashType}:$

$\text{ValidateMAC}(\text{LL1NVRAM}.\text{symmetricKey}', \text{historyStateBinding}, \text{LL1RAM}.\text{authenticator}') \Rightarrow$

$\text{LL1RAM}.\text{authenticator}' \in \text{LL1ObservedAuthenticators}'$

(4) 1. $\text{ExtendedUnforgeabilityInvariant} \land \text{LL1TypeInvariant} \land \text{LL1TypeInvariant}'$

By (2) 1

(4) 2. UNCHANGED LL1RAM

By (2) 2 DEF LL1Vars

(4) 3. UNCHANGED LL1ObservedAuthenticators

By (2) 2 DEF LL1Vars

(4) 4. UNCHANGED LL1NVRAM.\text{symmetricKey}

(5) 1. UNCHANGED LL1NVRAM

By (2) 2 DEF LL1Vars

(5) 2. QED

By (5) 1

(4) 5. QED

By (4) 1, (4) 2, (4) 3, (4) 4, LL1RAMUnforgeabilityUnchangedLemma

(3) 2. $\forall \text{historyStateBinding} \in \text{HashType} :$

$\text{ValidateMAC}(\text{LL1NVRAM}.\text{symmetricKey}', \text{historyStateBinding}, \text{LL1Disk}.\text{authenticator}') \Rightarrow$

$\text{LL1Disk}.\text{authenticator}' \in \text{LL1ObservedAuthenticators}'$

(4) 1. $\text{ExtendedUnforgeabilityInvariant} \land \text{LL1TypeInvariant} \land \text{LL1TypeInvariant}'$

By (2) 1

(4) 2. UNCHANGED LL1Disk

By (2) 2 DEF LL1Vars

(4) 3. UNCHANGED LL1ObservedAuthenticators
We break down the \( LL_2 \langle LL \rangle_3 \) case into eight separate cases for each action.

(2.3) \textbf{CASE \textit{LL1Next}}

The \textit{LL1MakeInputAvailable} case is a straightforward application of the \textit{LL1RAMUnforgeabilityUnchangedLemma} and the \textit{LL1DiskUnforgeabilityUnchangedLemma}.

(3.1) \textbf{CASE \textit{LL1MakeInputAvailable}}

(4.1) \textbf{PICK input } \in \textit{InputType} : \textit{LL1MakeInputAvailable}! (input)

(4.2) \forall \textit{historyStateBinding} \in \textit{HashType} :
\[
\text{ValidateMAC}(\textit{LL1NVRAM}.\textit{symmetricKey}', \textit{historyStateBinding}, \textit{LL1RAM}.\textit{authenticator}') \Rightarrow \textit{LL1RAM}.\textit{authenticator}' \in \textit{LL1ObservedAuthenticators}'
\]

(5.1) \textit{ExtendedUnforgeabilityInvariant} \wedge \textit{LL1TypeInvariant} \wedge \textit{LL1TypeInvariant}'

(5.2) \textbf{UNCHANGED \textit{LL1RAM}}

(5.3) \textbf{UNCHANGED \textit{LL1ObservedAuthenticators}}

(5.4) \textbf{UNCHANGED \textit{LL1NVRAM}.\textit{symmetricKey}}

(6.1) \textbf{UNCHANGED \textit{LL1NVRAM}}

(4.4.4) \textbf{UNCHANGED \textit{LL1Disk}}

(5.5) \textbf{QED}

(4.4.5) \textbf{QED}

The \textit{LL1PerformOperation} case is not terribly involved, but we have to treat the RAM and disk conjuncts separately.
For the disk portion of the unforgeability invariant, we employ the \( LL^1 \) PerformOperation action. For the RAM portion of the unforgeability invariant, we note that the action updates the authenticator in the RAM with the new authenticator. Since this new authenticator is unioned into the set of observed authenticators, the invariant holds in the primed state.

\( 3.2 \) CASE \( LL^1 \) PerformOperation

We pick some input for which \( LL^1 \) PerformOperation is true.

\( 4.1 \) PICK input \( \in LL^1 \) AvailableInputs : \( LL^1 \) PerformOperation!(input)!

BY \( 3.2 \) DEF \( LL^1 \) PerformOperation

To simplify the writing of the proof, we re-state the definitions from the \( LL^1 \) PerformOperation action.

\( 4. \) stateHash \( \triangleq \) Hash(\( LL^1 \) RAM.publicState, \( LL^1 \) RAM.privateStateEnc)

\( 4. \) historyStateBinding \( \triangleq \) Hash(\( LL^1 \) NVRAM.historySummary, stateHash)

\( 4. \) privateState \( \triangleq \) SymmetricDecrypt(\( LL^1 \) NVRAM.symmetricKey, \( LL^1 \) RAM.privateStateEnc)

\( 4. \) sResult \( \triangleq \) Service(\( LL^1 \) RAM.publicState, privateState, input)

\( 4. \) newPrivateStateEnc \( \triangleq \)

\[ \text{SymmetricEncrypt}(\( LL^1 \) NVRAM.symmetricKey, sResult.newPrivateState) \]

\( 4. \) newHistorySummary \( \triangleq \) Hash(\( LL^1 \) NVRAM.historySummary, input)

\( 4. \) newStateHash \( \triangleq \) Hash(sResult.newPublicState, newPrivateStateEnc)

\( 4. \) newHistoryStateBinding \( \triangleq \) Hash(newHistorySummary, newStateHash)

\( 4. \) newAuthenticator \( \triangleq \) GenerateMAC(\( LL^1 \) NVRAM.symmetricKey, newHistoryStateBinding)

We hide the definitions, so they don’t overwhelm the prover. We’ll pull them in as necessary below.

\( 4. \) USE DEF ExtendedUnforgeabilityInvariant

\( 4.2 \) TAKE historyStateBinding \( 1 \in \text{HashType} \)

Before proceeding to prove each of the conjuncts, we prove a statement that will be useful in both of the sub-proofs below. Namely, the new authenticator generated by the \( LL^1 \) PerformOperation action is unioned into the set of observed authenticators.

\( 4.3 \) \( LL^1 \) ObservedAuthenticators' = \( LL^1 \) ObservedAuthenticators \( \cup \) \{newAuthenticator\}

BY \( 4.1 \) DEF newAuthenticator, newHistoryStateBinding, newStateHash, newHistorySummary, newPrivateStateEnc, sResult, privateState

For the RAM portion of the unforgeability invariant, we note that the \( LL^1 \) PerformOperation action updates the authenticator in the RAM with the new authenticator. Since this new authenticator is unioned into the set of observed authenticators, the invariant holds in the primed state.

\( 4.4 \) ValidateMAC(\( LL^1 \) NVRAM.symmetricKey', historyStateBinding1, \( LL^1 \) RAM.authenticator') \( \Rightarrow \)

\( LL^1 \) RAM.authenticator' \( \in \) \( LL^1 \) ObservedAuthenticators'

\( 5.1 \) \( LL^1 \) RAM.authenticator' = newAuthenticator

\( 5.6 \) \( \text{newAuthenticator} = LL^1 \) PerformOperation!(input)!newAuthenticator

BY \( 4.1 \) DEF newAuthenticator, newHistoryStateBinding, newStateHash, newHistorySummary, newPrivateStateEnc, sResult, privateState

\( 5.6 \) \( LL^1 \) RAM' = \{ publicState \( \mapsto \) \( LL^1 \) PerformOperation!(input)!sResult.newPublicState, privateStateEnc \( \mapsto \) \( LL^1 \) PerformOperation!(input)!newPrivateStateEnc, historySummary \( \mapsto \) \( LL^1 \) PerformOperation!(input)!newHistorySummary, authenticator \( \mapsto \) newAuthenticator \}

BY \( 4.1 \), \( 6.1 \)

\( 6.3 \) QED

BY \( 6.2 \)

\( 5.2 \) QED

BY \( 4.3 \), \( 5.1 \)

For the disk portion of the unforgeability invariant, we employ the \( LL^1 \) DiskUnforgeabilityUnchangedLemma, since the disk is not changed by the \( LL^1 \) PerformOperation action.
The $LL1RepeatOperation$ case is not terribly involved, but we have to treat the RAM and disk conjuncts separately.

(3.3) CASE $LL1RepeatOperation$

We pick some input for which $LL1RepeatOperation$ is true.

(4.1) PICK $input \in LL1AvailableInputs : LL1RepeatOperation!(input)!1$

BY (3.3) DEF $LL1RepeatOperation$

To simplify the writing of the proof, we re-state the definitions from the $LL1RepeatOperation$ action.

(4) $stateHash \triangleq Hash(LL1RAM.publicState, LL1RAM.privateStateEnc)$
(4) $historyStateBinding \triangleq Hash(LL1RAM.historySummary, stateHash)$
(4) $privateState \triangleq SymmetricDecrypt(LL1NVRAM.symmetricKey, LL1RAM.privateStateEnc)$
(4) $sResult \triangleq Service(LL1RAM.publicState, privateState, input)$
(4) $newPrivateStateEnc \triangleq \text{SymmetricEncrypt}(LL1NVRAM.symmetricKey, sResult.newPrivateState)$
(4) $newStateHash \triangleq Hash(sResult.newPublicState, newPrivateStateEnc)$
(4) $newHistoryStateBinding \triangleq Hash(LL1NVRAM.historySummary, newStateHash)$
(4) $newAuthenticator \triangleq GenerateMAC(LL1NVRAM.symmetricKey, newHistoryStateBinding)$

We hide the definitions, so they don’t overwhelm the prover. We’ll pull them in as necessary below.

(4) HIDE DEF $stateHash, historyStateBinding, privateState, sResult, newPrivateStateEnc,$
$newStateHash, newHistoryStateBinding, newAuthenticator$

To prove the universally quantified expression, we take a new hash. For the $\text{TAKE}$ step to be meaningful to the prover, first we have to tell the prover to expand the definition of $ExtendedUnforgeabilityInvariant$, so it will see the universally quantified expression therein.

(4) USE DEF $ExtendedUnforgeabilityInvariant$
(4) $\text{TAKE} historyStateBinding1 \in HashType$

Before proceeding to prove each of the conjuncts, we prove a statement that will be useful in both of the sub-proofs below. Namely, the new authenticator generated by the $LL1PerformOperation$ action is unioned into the set of observed authenticators.

(4.3) $LL1ObservedAuthenticators' = LL1ObservedAuthenticators \cup \{\text{newAuthenticator}\}$

BY (4.1) DEF $\text{newAuthenticator}, newHistoryStateBinding, newStateHash, newPrivateStateEnc, sResult, privateState$
For the RAM portion of the unforgeability invariant, we note that the \( LL1\text{RepeatOperation} \) action updates the authenticator in the RAM with the new authenticator. Since this new authenticator is unioned into the set of observed authenticators, the invariant holds in the primed state.

(4.4). \( \text{ValidateMAC}(LL1\text{NVRAM}.\text{symmetricKey}', \text{historyStateBinding1}, LL1\text{RAM}.\text{authenticator}') \Rightarrow LL1\text{RAM}.\text{authenticator}' \in LL1\text{ObservedAuthenticators}'

(5.1). \( LL1\text{RAM}.\text{authenticator}' = \text{newAuthenticator} \)

(6.1). \( \text{newAuthenticator} = LL1\text{RepeatOperation}!(\text{input})!\text{newAuthenticator} \)

BY (4.1) \( \text{DEF newAuthenticator, newHistoryStateBinding, newStateHash, } \\
\text{newPrivateStateEnc, sResult, privateState} \)

(6.2). \( LL1\text{RAM}' = [ \\
\text{publicState} \mapsto LL1\text{RepeatOperation}!(\text{input})!\text{sResult}.\text{newPublicState}, \\
\text{privateStateEnc} \mapsto LL1\text{RepeatOperation}!(\text{input})!\text{newPrivateStateEnc}, \\
\text{historySummary} \mapsto LL1\text{NVRAM}.\text{historySummary}, \\
\text{authenticator} \mapsto \text{newAuthenticator}] \)

BY (4.1), (6.1)

(6.3). QED

BY (6.2)

(5.2). QED

BY (4.3), (5.1)

For the disk portion of the unforgeability invariant, we employ the \( LL1\text{DiskUnforgeabilityUnchangedLemma} \), since the disk is not changed by the \( LL1\text{RepeatOperation} \) action.

(4.5). \( \text{ValidateMAC}(LL1\text{NVRAM}.\text{symmetricKey}', \text{historyStateBinding1}, LL1\text{Disk}.\text{authenticator}') \Rightarrow \\
LL1\text{Disk}.\text{authenticator}' \in LL1\text{ObservedAuthenticators}' \)

(5.1). \( \text{ExtendedUnforgeabilityInvariant} \land LL1\text{TypeInvariant} \land LL1\text{TypeInvariant}' \)

BY (2.1)

(5.2). UNCHANGED \( LL1\text{Disk} \)

BY (4.1)

(5.3). \( LL1\text{ObservedAuthenticators} \subseteq LL1\text{ObservedAuthenticators}' \)

BY (4.3)

(5.4). UNCHANGED \( LL1\text{NVRAM}.\text{symmetricKey} \)

(6.1). UNCHANGED \( LL1\text{NVRAM} \)

BY (4.1)

(6.2). QED

BY (6.1)

(5.5). QED

BY (5.1), (5.2), (5.3), (5.4), \( LL1\text{RAMUnforgeabilityUnchangedLemma} \)

(4.6). QED

BY (4.4), (4.5)

The \( LL1\text{Restart} \) case is moderately interesting. The RAM portion of the unforgeability invariant is where we pull in the constraint on the set of authenticators that can wind up in the randomized RAM after a \( LL1\text{Restart} \). This constraint is explicitly expressed in the definition of the \( LL1\text{Restart} \) action. The disk portion is a straightforward application of the \( LL1\text{DiskUnforgeabilityUnchangedLemma} \).

(3.4). CASE \( LL1\text{Restart} \)

We pick some variables of the appropriate types for which \( LL1\text{Restart} \) is true.

(4.1). PICK \( \text{untrustedStorage} \in LL1\text{UntrustedStorageType}, \\
\text{randomSymmetricKey} \in \text{SymmetricKeyType} \setminus \{ LL1\text{NVRAM}.\text{symmetricKey} \}, \\
\text{hash} \in \text{HashType} : \\
LL1\text{Restart}!(\text{untrustedStorage}, \text{randomSymmetricKey}, \text{hash}) \)

BY (3.4) \( \text{DEF LL1\text{Restart}} \)

To prove the universally quantified expression, we take a new hash. For the \text{take} step to be meaningful to the prover, first we have to tell the prover to expand the definition of \( \text{ExtendedUnforgeabilityInvariant} \), so it will see the universally quantified expression therein.
For the RAM portion, we first note that the definition of $LL1Restart$ tells us that the primed authenticator has been constructed with a random symmetric key. The antecedent of the implication cannot be true, because the random symmetric key does not match the symmetric key in the NVRAM. We show this using proof by contradiction.

(5.1) $LL1RAM.authenticator' = \text{GenerateMAC}(randomSymmetricKey, hash)$

BY (4)1

(5.2) $\neg \text{ValidateMAC}(LL1NVRAM.symmetricKey', historyStateBinding, LL1RAM.authenticator')$

(6.1) SUFFICES

ASSUME

\text{ValidateMAC}(
    LL1NVRAM.symmetricKey',
    historyStateBinding,
    LL1RAM.authenticator')

PROVE FALSE

BY (5)1

(6.2) randomSymmetricKey = LL1NVRAM.symmetricKey'

(7.1) ValidateMAC(
    LL1NVRAM.symmetricKey',
    historyStateBinding,
    GenerateMAC(randomSymmetricKey, hash))

BY (5)1, (6)1

(7.2) randomSymmetricKey \in \text{SymmetricKeyType}

BY (4)1

(7.3) LL1NVRAM.symmetricKey' \in \text{SymmetricKeyType}

(8.1) $LL1TypeInvariant'$

BY (2)1

(8.2) QED

BY (8)1, $LL1SubtypeImplicationLemma$ Defined $LL1SubtypeImplication$

(7.4) hash \in \text{HashType}

BY (4)1

(7.5) historyStateBinding \in \text{HashType}

BY (4)3

(7.6) QED

BY (7)1, (7)2, (7)3, (7)4, (7)5, MACUnforgeable

(6.3) randomSymmetricKey \neq LL1NVRAM.symmetricKey'

(7.1) randomSymmetricKey \neq LL1NVRAM.symmetricKey

BY (4)1

(7.2) UNCHANGED LL1NVRAM.symmetricKey

BY (2)3, $\text{SymmetricKeyConstantLemma}$

(7.3) QED

BY (7)1, (7)2

(6.4) QED

BY (6)2, (6)3

(5.3) QED

BY (5)2

For the disk portion of the unforgeability invariant, we employ the $LL1DiskUnforgeabilityUnchangedLemma$, since the disk is not changed by the $LL1CorruptRAM$ action.

(4.5) ValidateMAC(LL1NVRAM.symmetricKey', historyStateBinding, LL1Disk.authenticator') \Rightarrow
\( \text{LL1Disk}.\text{decryptor}^* \in \text{LL1ObservedAuthenticators}' \)

(5.1) \( \text{ExtendedUnforgeabilityInvariant} \land \text{LL1TypeInvariant} \land \text{LL1TypeInvariant}' \)

by (2)1

(5.2) UNCHANGED LL1Disk

by (4)1

(5.3) UNCHANGED LL1ObservedAuthenticators

by (4)1

(5.4) UNCHANGED LL1NVRAM.\text{symmetricKey}

(6.1) UNCHANGED LL1NVRAM

by (4)1

(6.2) QED

by (6)1

(5.5) QED

by (5)1, (5)2, (5)3, (5)4, LL1DiskUnforgeabilityUnchangedLemma

(4)6. QED

by (4)4, (4)5

The LL1ReadDisk case is a straightforward application of the LL1RAMUnforgeabilityUnchangedLemma and the LL1DiskUnforgeabilityUnchangedLemma.

(3)5. CASE LL1ReadDisk

(4.1) \( \forall \text{historyStateBinding} \in \text{HashType} : \)

\( \text{ValidateMAC}(\text{LL1NVRAM.\text{symmetricKey}'} \land \text{historyStateBinding}, \text{LL1RAM}.\text{decryptor}^*) \Rightarrow \)

\( \text{LL1RAM}.\text{decryptor}^* \in \text{LL1ObservedAuthenticators}' \)

(5.1) ExtendedUnforgeabilityInvariant \land \text{LL1TypeInvariant} \land \text{LL1TypeInvariant}'

by (2)1

(5.2) LL1RAM' = LL1Disk

by (3)5 DEF LL1ReadDisk

(5.3) UNCHANGED LL1ObservedAuthenticators

by (3)5 DEF LL1ReadDisk

(5.4) UNCHANGED LL1NVRAM.\text{symmetricKey}

(6.1) UNCHANGED LL1NVRAM

by (3)5 DEF LL1ReadDisk

(6.2) QED

by (6)1

(5.5) QED

by (5)1, (5)2, (5)3, (5)4, LL1RAMUnforgeabilityUnchanged Lemma

(4)2. \( \forall \text{historyStateBinding} \in \text{HashType} : \)

\( \text{ValidateMAC}(\text{LL1NVRAM.\text{symmetricKey}'} \land \text{historyStateBinding}, \text{LL1Disk}.\text{decryptor}^*) \Rightarrow \)

\( \text{LL1Disk}.\text{decryptor}^* \in \text{LL1ObservedAuthenticators}' \)

(5.1) ExtendedUnforgeabilityInvariant \land \text{LL1TypeInvariant} \land \text{LL1TypeInvariant}'

by (2)1

(5.2) UNCHANGED LL1Disk

by (3)5 DEF LL1ReadDisk

(5.3) UNCHANGED LL1ObservedAuthenticators

by (3)5 DEF LL1ReadDisk

(5.4) UNCHANGED LL1NVRAM.\text{symmetricKey}

(6.1) UNCHANGED LL1NVRAM

by (3)5 DEF LL1ReadDisk

(6.2) QED

by (6)1

(5.5) QED

by (5)1, (5)2, (5)3, (5)4, LL1DiskUnforgeabilityUnchangedLemma

(4)3. QED
BY (4)1, (4)2 DEF ExtendedUnforgeabilityInvariant

The LL1WriteDisk case is a straightforward application of the LL1RAMUnforgeabilityUnchangedLemma and the LL1DiskUnforgeabilityUnchangedLemma.

(3)6. CASE LL1WriteDisk

(4)1. ∀ historyStateBinding ∈ HashType :

    ValidateMAC(LL1NVRAM.symmetricKey', historyStateBinding, LL1RAM.authenticate') ⇒
    LL1RAM.authenticate' ∈ LL1ObservedAuthenticators'

(5)1. ExtendedUnforgeabilityInvariant ∧ LL1TypeInvariant ∧ LL1TypeInvariant'

    BY (2)1

(5)2. UNCHANGED LL1RAM

    BY (3)6 DEF LL1WriteDisk

(5)3. UNCHANGED LL1ObservedAuthenticators

    BY (3)6 DEF LL1WriteDisk

(5)4. UNCHANGED LL1NVRAM.symmetricKey

    (6)1. UNCHANGED LL1NVRAM

    BY (3)6 DEF LL1WriteDisk

    (6)2. QED

    BY (6)1

(5)5. QED

    BY (5)1, (5)2, (5)3, (5)4, LL1RAMUnforgeabilityUnchangedLemma

(4)2. ∀ historyStateBinding ∈ HashType :

    ValidateMAC(LL1NVRAM.symmetricKey', historyStateBinding, LL1Disk.authenticate') ⇒
    LL1Disk.authenticate' ∈ LL1ObservedAuthenticators'

(5)1. ExtendedUnforgeabilityInvariant ∧ LL1TypeInvariant ∧ LL1TypeInvariant'

    BY (2)1

(5)2. LL1Disk' = LL1RAM

    BY (3)6 DEF LL1WriteDisk

(5)3. UNCHANGED LL1ObservedAuthenticators

    BY (3)6 DEF LL1WriteDisk

(5)4. UNCHANGED LL1NVRAM.symmetricKey

    (6)1. UNCHANGED LL1NVRAM

    BY (3)6 DEF LL1WriteDisk

    (6)2. QED

    BY (6)1

(5)5. QED

    BY (5)1, (5)2, (5)3, (5)4, LL1DiskUnforgeabilityUnchangedLemma

(4)3. QED

    BY (4)1, (4)2 DEF ExtendedUnforgeabilityInvariant

The LL1CorruptRAM case is moderately interesting. The RAM portion of the unforgeability invariant is where we pull in the constraint on the set of authenticators the user can create. This constraint is explicitly expressed in the definition of the LL1CorruptRAM action. The disk portion is a straightforward application of the LL1DiskUnforgeabilityUnchangedLemma.

(3)7. CASE LL1CorruptRAM

We pick some variables of the appropriate types for which LL1CorruptRAM is true.

(4)1. PICK untrustedStorage ∈ LL1UntrustedStorageType,

    fakeSymmetricKey ∈ SymmetricKeyType \ {LL1NVRAM.symmetricKey},

    hash ∈ HashType :

    LL1CorruptRAM!(untrustedStorage, fakeSymmetricKey, hash)

    BY (3)7 DEF LL1CorruptRAM

To prove the universally quantified expression, we take a new hash. For the take step to be meaningful to the prover, first we have to tell the prover to expand the definition of ExtendedUnforgeabilityInvariant, so it will see the universally quantified expression therein.
(4) USE DEF ExtendedUnforgeabilityInvariant

(4.3) TAKE historyStateBinding ∈ HashType
(4.4) ValidateMAC(LL1NVRAM.symmetricKey', historyStateBinding, LL1RAM.authenticator') ⇒
    LL1RAM.authenticator' ∈ LL1ObservedAuthenticators'

For the RAM portion, we first note that the definition of LL1CorruptRAM tells us that the primed authenticator in
the RAM is either in the unprimed set of observed authenticators or has been constructed with a fake
symmetric key. We will treat these two cases separately.

(5.1) ∨ LL1RAM.authenticator' ∈ LL1ObservedAuthenticators
    ∨ LL1RAM.authenticator' = GenerateMAC(fakeSymmetricKey, hash)
    BY (4.1)

For the case in which the primed authenticator is in the unprimed set of observed authenticators, the conclusion
directly follows because the set of observed authenticators is not changed by the LL1CorruptRAM action.

(5.2) CASE LL1RAM.authenticator' ∈ LL1ObservedAuthenticators
    (6.1) HAVE ValidateMAC(
            LL1NVRAM.symmetricKey', historyStateBinding, LL1RAM.authenticator')
    (6.2) UNCHANGED LL1ObservedAuthenticators
        BY (4.1)
    (6.3) QED
        BY (5.2), (6.2)

For the case in which the primed authenticator has been constructed with a fake symmetric key, the antecedent
of the implication cannot be true, because the fake symmetric key does not match the symmetric key in the
NVRAM. We show this using proof by contradiction.

(5.3) CASE LL1RAM.authenticator' = GenerateMAC(fakeSymmetricKey, hash)
    (6.1) ¬ValidateMAC(LL1NVRAM.symmetricKey', historyStateBinding, LL1RAM.authenticator')
        SUFFICES
        ASSUME
        ValidateMAC(
            LL1NVRAM.symmetricKey',
            historyStateBinding,
            LL1RAM.authenticator')
        PROVE FALSE
        BY (5.3)
    (7.1) fakeSymmetricKey = LL1NVRAM.symmetricKey'
        (8.1) ValidateMAC(
                LL1NVRAM.symmetricKey',
                historyStateBinding,
                GenerateMAC(fakeSymmetricKey, hash))
            BY (5.3), (7.1)
        (8.2) fakeSymmetricKey ∈ SymmetricKeyType
            BY (4.1)
        (8.3) LL1NVRAM.symmetricKey' ∈ SymmetricKeyType
            (9.1) LL1TypeInvariant'
                BY (2.1)
            (9.2) QED
                BY (9.1), LL1SubtypeImplicationLemma DEF LL1SubtypeImplication
        (8.4) hash ∈ HashType
            BY (4.1)
        (8.5) historyStateBinding ∈ HashType
            BY (4.3)
        (8.6) QED
            BY (8.1), (8.2), (8.3), (8.4), (8.5), MACUnforgeable
    (7.3) fakeSymmetricKey ≠ LL1NVRAM.symmetricKey'

116
THE WIND UP IN THE RANDOMIZED RAM AFTER THE ACTION COMPLETES. THE DISK PORTION IS A STRAIGHTFORWARD APPLICATION OF LL ON WHETHER THE LL.

3.

TO PROVE THE UNIVERSALLY QUANTIFIED EXPRESSION, WE TAKE A NEW HASH. FOR THE DISK PORTION OF THE UNFORGEABILITY INVARIANT, WE EMPLOY THE ExtendedUnforgeabilityInvariant USE DEF.

4.

THE LL1RestrictedCorruption CASE IS MODERATELY INTERESTING. FOR THE RAM PORTION, WE HAVE TWO CASES DEPENDING ON WHETHER THE LL1RestrictedCorruption ACTION LEAVES THE RAM UNCHANGED OR TRASHES IT IN THE SAME WAY AN LL1Restart ACTION DOES. FOR THE LATTER CASE, WE PULL IN THE CONSTRAINT ON THE SET OF STATE AUTHENTICATORS THAT CAN WIND UP IN THE RANDOMIZED RAM AFTER THE ACTION COMPLETES. THE DISK PORTION IS A STRAIGHTFORWARD APPLICATION OF THE LL1DiskUnforgeabilityUnchangedLemma.

5.

(5) CASE LL1RestrictedCorruption

To prove the universally quantified expression, we take a new hash. For the take step to be meaningful to the prover, first we have to tell the prover to expand the definition of ExtendedUnforgeabilityInvariant, so it will see the universally quantified expression therein.

6.

(5) CASE LL1RestrictedCorruption!ram!unchanged

First we consider the case in which the RAM is unchanged. This is straightforward

7.

(5)1. CASE LL1RestrictedCorruption!ram!unchanged

(5) CASE LL1RestrictedCorruption

(5) USE DEF ExtendedUnforgeabilityInvariant

(4)1. TAKE historyStateBinding ∈ HashType

(4)2. ValidateMAC(LL1NVRAM.symmetricKey', historyStateBinding, LL1Disk.authenticator') ⇒ LL1Disk. authenticator' ∈ LL1ObservedAuthenticators'
Next we consider the case in which the RAM is trashed.

We pick some variables of the appropriate types for which \( LL_1\) \text{RestrictedCorruption}! \( \text{ram!trashed} \) is true.

The primed authenticator has been constructed with a random symmetric key. The antecedent of the implication cannot be true, because the random symmetric key does not match the symmetric key in the \( NVRAM \).

We show this using proof by contradiction.

The suffices assume

\[
\begin{align*}
\text{ValidateMAC}( & \quad LL_1NVRAM.\text{symmetricKey}', \\
& \quad \text{historyStateBinding}, \\
& \quad LL_1\text{RAM.authenticator}')
\end{align*}
\]

Prove false

by (6)1

\( 8.1 \) randomSymmetricKey = \( LL_1NVRAM.\text{symmetricKey}' \)

by (6)1, (7)1

\( 8.2 \) randomSymmetricKey \( \in \) SymmetricKeyType

by (6)1

\( 8.3 \) \( LL_1NVRAM.\text{symmetricKey}' \) \( \in \) SymmetricKeyType

by (2)1

\( 8.4 \) hash \( \in \) HashType

by (6)1

\( 8.5 \) historyStateBinding \( \in \) HashType

by (6)3

\( 8.6 \) QED

by (8)1, (8)2, (8)3, (8)4, (8)5, MACUnforgeable

(7)3. randomSymmetricKey \( \neq \) \( LL_1NVRAM.\text{symmetricKey}' \)

by (6)1

\( 8.1 \) randomSymmetricKey \( \neq \) \( LL_1NVRAM.\text{symmetricKey} \)

by (6)1

\( 8.2 \) UNCHANGED \( LL_1NVRAM.\text{symmetricKey} \)

by (2)3, SymmetricKeyConstantLemma

\( 8.3 \) QED
For the disk portion of the unforgeability invariant, we employ the $LL_1\text{DiskUnforgeabilityUnchangedLemma}$, since the disk is not changed by the $LL_1\text{CorruptRAM}$ action.

(4.3). ValidateMAC($LL_1\text{NVRAM}.\text{symmetricKey}'$, historyStateBinding, $LL_1\text{Disk}.\text{authenticator}'$) ⇒ $LL_1\text{Disk}.\text{authenticator}' \in LL_1\text{ObservedAuthenticators}'$

(5.1). $\text{ExtendedUnforgeabilityInvariant} \land LL_1\text{TypeInvariant} \land LL_1\text{TypeInvariant}'$

BY (2.1)

(5.2). UNCHANGED $LL_1\text{Disk}$

BY (3.8) DEF $LL_1\text{RestrictedCorruption}$

(5.3). UNCHANGED $LL_1\text{ObservedAuthenticators}$

BY (3.8) DEF $LL_1\text{RestrictedCorruption}$

(5.4). UNCHANGED $LL_1\text{NVRAM}.\text{symmetricKey}$

BY (2.3), $\text{SymmetricKeyConstantLemma}$

(5.5). QED

BY (5.1), (5.2), (5.3), (5.4), $LL_1\text{DiskUnforgeabilityUnchangedLemma}$

(4.4). QED

BY (4.2), (4.3)

(3.9). QED

BY (2.3), (3.1), (3.2), (3.3), (3.4), (3.5), (3.6), (3.7), (3.8) DEF $LL_1\text{Next}$

(2.4). QED

BY (2.1), (2.2), (2.3)

(1.4). QED

Using the Inv1 proof rule, the base case and the induction step together imply that the invariant always holds.

(2.1). ( ∧ $\text{ExtendedUnforgeabilityInvariant}$ ∧ $\Box[LL_1\text{Next}]_{LL_1\text{Vars}}$ ∧ $\Box LL_1\text{TypeInvariant}$) ⇒

$\Box\text{ExtendedUnforgeabilityInvariant}$

BY (1.3), Inv1

(2.2). QED

BY (1.1), (1.2) DEF $LL_1\text{Spec}$

The UnforgeabilityInvariant follows directly from the ExtendedUnforgeabilityInvariant.

THEOREM UnforgeabilityInvariance $\triangleq LL_1\text{Spec} \Rightarrow \Box \text{UnforgeabilityInvariant}$

(1.1). $LL_1\text{Spec} \Rightarrow \Box \text{ExtendedUnforgeabilityInvariant}$

BY ExtendedUnforgeabilityInvariance

(1.2). $\text{ExtendedUnforgeabilityInvariant} \Rightarrow \text{UnforgeabilityInvariant}$

BY DEF $\text{ExtendedUnforgeabilityInvariant}$, $\text{UnforgeabilityInvariant}$

(1.3). QED

BY (1.1), (1.2)
4.6 Proof of Inclusion, Cardinality, and Uniqueness Co-invariance in Memoir-Basic

This module proves that the InclusionInvariant, the CardinalityInvariant, and the UniquenessInvariant are all inductive invariants of the Memoir-Basic spec. Then, from this proof and the previous proof that the UnforgeabilityInvariant is an inductive invariant of the Memoir-Basic spec, it proves that the set of CorrectnessInvariants are inductive invariants of the Memoir-Basic spec.

EXTENDS MemoirLL1 UnforgeabilityInvariant

These three invariants cannot be ordered with respect to each other. The proof of InclusionInvariant inductively depends upon UniquenessInvariant; the proof of UniquenessInvariant inductively depends upon CardinalityInvariant; and the proof of CardinalityInvariant inductively depends upon InclusionInvariant. Therefore, we have to prove them co-inductively.

THEOREM InclusionCardinalityUniquenessInvariance  \[\Delta\]
\[LL1\text{Spec} \Rightarrow \Box \text{InclusionInvariant} \land \Box \text{CardinalityInvariant} \land \Box \text{UniquenessInvariant}\]

This proof will require the LL1 TypeInvariant and the UnforgeabilityInvariant. Fortunately, the LL1 TypeSafe theorem has already proven that the Memoir-Basic spec satisfies its type invariant, and the UnforgeabilityInvariant theorem has already proven that the Memoir-Basic spec satisfies the UnforgeabilityInvariant.

⟨1⟩. \[LL1\text{Spec} \Rightarrow \Box \text{LL1 TypeInvariant}\]
   BY LL1 TypeSafe

⟨1⟩. \[LL1\text{Spec} \Rightarrow \Box \text{UnforgeabilityInvariant}\]
   BY UnforgeabilityInvariant

The top level of the proof is boilerplate TLA+ for an Inv1-style proof. First, we prove that the initial state satisfies the three invariants. Second, we prove that the LL1Next predicate inductively preserves the three invariants. Third, we use temporal induction to prove that these two conditions satisfy the three invariants over all behaviors.

⟨1⟩. \[LL1\text{Init} \land LL1\text{TypeInvariant} \land \text{UnforgeabilityInvariant} \Rightarrow \text{InclusionInvariant} \land \text{CardinalityInvariant} \land \text{UniquenessInvariant}\]

First, we assume the antecedents.

⟨2⟩. \text{HAVE LL1 Init} \land LL1 TypeInvariant

Then, we pick some symmetric key for which LL1 Init is true.

⟨2⟩. \text{PICK symmetricKey} \in \text{SymmetricKeyType} : LL1 Init!(symmetricKey)!1
   BY ⟨2⟩ DEF LL1 Init

To simplify the writing of the proof, we re-state the definitions from LL1 Init.

⟨2⟩. \text{initialPrivateStateEnc} \triangleq \text{SymmetricEncrypt}(\text{symmetricKey}, \text{InitialPrivateState})
⟨2⟩. \text{initialStateHash} \triangleq \text{Hash}(\text{InitialPublicState}, \text{initialPrivateStateEnc})
⟨2⟩. \text{initialHistoryStateBinding} \triangleq \text{Hash}(\text{BaseHashValue}, \text{initialStateHash})
⟨2⟩. \text{initialAuthenticator} \triangleq \text{GenerateMAC}(\text{symmetricKey}, \text{initialHistoryStateBinding})
⟨2⟩. \text{initialUntrustedStorage} \triangleq [\text{initialTrustedStorage} \triangleq [\text{historySummary} \mapsto \text{BaseHashValue}, \text{symmetricKey} \mapsto \text{symmetricKey}]]

We then assert the type safety of these definitions, with the help of the LL1 InitDefsTypeSafeLemma.

⟨2⟩. \land \text{initialPrivateStateEnc} \in \text{PrivateStateEncType}
   \land \text{initialStateHash} \in \text{HashType}
   \land \text{initialHistoryStateBinding} \in \text{HashType}
   \land \text{initialAuthenticator} \in \text{MACType}
initialUntrustedStorage ∈ LL₁UntrustedStorageType
∧ initialTrustedStorage ∈ LL₁TrustedStorageType
(3.1). symmetricKey ∈ SymmetricKeyType
   BY (2)2
(3.2). QED
   BY (3.1), LL₁InitDefsTypeSafeLemma

We hide the definitions, so they don’t overwhelm the prover. We’ll pull them in as necessary below.

(2.2) HIDE DEF initialPrivateStateEnc, initialStateHash, initialHistoryStateBinding,
   initialAuthenticator, initialUntrustedStorage, initialTrustedStorage

We’ll prove each of the three invariants separately. First, we’ll prove the InclusionInvariant.

(2.4). InclusionInvariant

To prove the universally quantified expression, we take a new set of variables of the appropriate types. For the take
step to be meaningful to the prover, first we have to tell the prover to expand the definition of InclusionInvariant,
so it will see the universally quantified expression therein.

(3) USE DEF InclusionInvariant
(3.1). TAKE input ∈ InputType,
   historySummary ∈ HashType,
   publicState ∈ PublicStateType,
   privateStateEnc ∈ PrivateStateEncType

It suffices to prove that the the antecedent of the implication is false, which merely requires showing that one of the
conjuncts is false. The proof is straightforward, since the history summary in the NVRAM equals the base hash
value, and the base hash value cannot be constructed as the hash of any other values.

(3.2). SUFFICES ASSUME TRUEPROVE LL₁NVRAM.historySummary ≠ Hash(historySummary, input)
      OBVIOUS
(3.3). LL₁NVRAM.historySummary = BaseHashValue
   (4.1). LL₁NVRAM = [historySummary → BaseHashValue,
   symmetricKey → symmetricKey]
      BY (2)2
   (4.2). QED
      BY (4.1)
(3.4). historySummary ∈ HashDomain
   (4.1). historySummary ∈ HashType
      BY (3)1
   (4.2). QED
      BY (4.1) DEF HashDomain
(3.5). input ∈ HashDomain
   (4.1). input ∈ InputType
      BY (3)1
   (4.2). QED
      BY (4.1) DEF HashDomain
(3.6). QED
      BY (3)3, (3.4), (3.5), BaseHashValueUnique

Second, we’ll prove the CardinalityInvariant.

(2.5). CardinalityInvariant

To prove the universally quantified expression, we take a new set of variables of the appropriate types. For the take
step to be meaningful to the prover, first we have to tell the prover to expand the definition of CardinalityInvariant,
so it will see the universally quantified expression therein.

(3) USE DEF CardinalityInvariant
(3.1). TAKE historySummary ∈ HashType, stateHash ∈ HashType

To simplify the writing of the proof, we re-state the definition from the CardinalityInvariant.

(3) historyStateBinding ≜ Hash(historySummary, stateHash)
We then assert the type safety of this definition, with the help of the \textit{CardinalityInvariantDefsTypeSafeLemma}.

\textbf{(3).2.} \texttt{historyStateBinding} \in \texttt{HashType}
\texttt{BY (3).1, CardinalityInvariantDefsTypeSafeLemma}

To prove the implication, it suffices to assume the antecedent and prove the consequent.

\textbf{(3).3. SUFFICES}
\textbf{ASSUME}
\begin{itemize}
\item \texttt{LL1NVRAMHistorySummaryUncorrupted}
\item \texttt{LL1HistoryStateBindingAuthenticated(historyStateBinding)}
\end{itemize}
\textbf{PROVE}
\texttt{HashCardinality(historySummary) \leq HashCardinality(LL1NVRAM.historySummary)}
\textbf{OBIous}

We hide the definition, so it doesn’t overwhelm the prover. We’ll pull it in as necessary below.

\textbf{(3) HIDE DEF \texttt{historyStateBinding}}

The proof is simple. First, we prove that the hash cardinality of the history summary is zero.

\textbf{(3).4.} \texttt{HashCardinality(historySummary) = 0}

There are two steps to proving that the hash cardinality of the history summary is zero. First, we use the \textit{MACCollisionResistant} property to prove that the history state binding matches the initial history state binding defined in \texttt{LL1Init}.

\textbf{(4).2.} \texttt{historyStateBinding} = \texttt{initialHistoryStateBinding}
\begin{itemize}
\item \texttt{ValidateMAC(LL1NVRAM.symmetricKey, historyStateBinding, initialAuthenticator)}
\item \texttt{BY (2).2 DEF \texttt{initialAuthenticator, initialHistoryStateBinding, initialStateHash, initialPrivateStateEnc}}
\item \texttt{BY (3).3}
\item \texttt{QED}
\item \texttt{BY (6).1, (6).2 DEF \texttt{LL1HistoryStateBindingAuthenticated}}
\end{itemize}
\texttt{\textbf{(5).2. LL1NVRAM.symmetricKey} \in \texttt{SymmetricKeyType}}
\begin{itemize}
\item \texttt{BY (2).1}
\item \texttt{QED}
\item \texttt{BY (6).1, LL1SubtypeImplicationLemma DEF LL1SubtypeImplication}
\item \texttt{\textbf{(5).3. symmetricKey} \in \texttt{SymmetricKeyType}}
\item \texttt{BY (2).2}
\item \texttt{\textbf{(5).4. historyStateBinding} \in \texttt{HashType}}
\item \texttt{BY (3).2}
\item \texttt{\textbf{(5).5. initialHistoryStateBinding} \in \texttt{HashType}}
\item \texttt{BY (2).3}
\item \texttt{QED}
\item \texttt{BY (5).1, (5).2, (5).3, (5).4, (5).5, MACCollisionResistant}
\item \texttt{DEF initialAuthenticator}
\end{itemize}

Second, we use the \textit{HashCollisionResistant} property to prove that the history summary matches the \texttt{BaseHashValue}, which is the initial history summary in \texttt{LL1Init}.

\textbf{(4).3.} \texttt{historySummary} = \texttt{BaseHashValue}
\begin{itemize}
\item \texttt{\textbf{(5).1. historySummary} \in \texttt{HashDomain}}
\item \texttt{BY (3).1}
\item \texttt{QED}
\item \texttt{BY (6).1 DEF \texttt{HashDomain}}
\item \texttt{\textbf{(5).2. stateHash} \in \texttt{HashDomain}}
\item \texttt{BY (6).1}
\end{itemize}

122
The conclusion follows directly, because the BaseHashValue has cardinality zero.

Then, we prove that the hash cardinality of the history summary in the NVRAM is zero.

Third, we’ll prove the UniquenessInvariant.

To prove the universally quantified expression, we take a new set of variables of the appropriate types. For the take step to be meaningful to the prover, first we have to tell the prover to expand the definition of UniquenessInvariant, so it will see the universally quantified expression therein.

We then assert the type safety of these definitions, with the help of the UniquenessInvariantDefsTypeSafeLemma.

To prove the implication, it suffices to assume the antecedent and prove the consequent.

ASSUME \( \wedge \) LL1HistoryStateBindingAuthenticated(historyStateBinding1) 
\( \wedge \) LL1HistoryStateBindingAuthenticated(historyStateBinding2)
We hide the definitions, so they don’t overwhelm the prover. We’ll pull them in as necessary below.

\(\text{prove } \text{stateHash}_1 = \text{stateHash}_2\)

\(\text{obvious}\)

Then, we’ll prove that the two history state bindings are equal to each other, because each one is equal to the initial state binding.

\(\text{1. } \text{historyStateBinding}_1 \text{ = historyStateBinding}_2\)

To prove that history state binding 1 equals the initial history state binding defined in \(\text{LL}_1\text{Init}\), we’ll appeal to the \text{MACCollisionResistant} property.

\(\text{5.1. } \text{ValidateMAC}(\text{LL}_1\text{NVRAM},\text{symmetricKey},\text{historyStateBinding}_1,\text{initialAuthenticator})\)

\(\text{6.1. } \text{LL}_1\text{ObservedAuthenticators} = \{\text{initialAuthenticator}\}\)

\(\text{by } \langle 2\rangle 2 \text{ def } \text{initialAuthenticator}, \text{initialHistoryStateBinding}, \text{initialStateHash}, \text{initialPrivateStateEnc}\)

\(\text{6.2. } \text{LL}_1\text{HistoryStateBindingAuthenticated}(\text{historyStateBinding}_1)\)

\(\text{by } \langle 3\rangle 3\)

\(\text{6.3. QED}\)

\(\text{by } \langle 6\rangle 1, \langle 6\rangle 2 \text{ def } \text{LL}_1\text{HistoryStateBindingAuthenticated}\)

We also have to prove the appropriate types, which are the next four statements.

\(\text{5.2. } \text{LL}_1\text{NVRAM}.\text{symmetricKey} \in \text{SymmetricKeyType}\)

\(\text{6.1. } \text{LL}_1\text{TypeInvariant}\)

\(\text{by } \langle 2\rangle 1\)

\(\text{6.2. QED}\)

\(\text{by } \langle 6\rangle 1, \text{LL}_1\text{SubtypeImplicationLemma}\text{ def } \text{LL}_1\text{SubtypeImplication}\)

\(\text{5.3. } \text{symmetricKey} \in \text{SymmetricKeyType}\)

\(\text{by } \langle 2\rangle 2\)

\(\text{5.4. } \text{historyStateBinding}_1 \in \text{HashType}\)

\(\text{by } \langle 3\rangle 2\)

\(\text{5.5. } \text{initialHistoryStateBinding} \in \text{HashType}\)

\(\text{by } \langle 2\rangle 3\)

Now, we can apply the \text{MACCollisionResistant} property, which establishes that history state binding 1 equals the initial history state binding.

\(\text{5.6. QED}\)

\(\text{by } \langle 5\rangle 1, \langle 5\rangle 2, \langle 5\rangle 3, \langle 5\rangle 4, \langle 5\rangle 5, \text{MACCollisionResistant}\text{ def } \text{initialAuthenticator}\)

To prove that history state binding 2 equals the initial history state binding defined in \(\text{LL}_1\text{Init}\), we’ll appeal to the \text{MACCollisionResistant} property. This is the exact same logic we followed above for history state binding 1.

\(\text{4.2. } \text{historyStateBinding}_2 \text{ = initialHistoryStateBinding}\)

The main precondition for \text{MACCollisionResistant} is that \text{historyStateBinding}_1 is validated by an authenticator that was generated with \text{initialHistoryStateBinding}.

\(\text{5.1. } \text{ValidateMAC}(\text{LL}_1\text{NVRAM},\text{symmetricKey},\text{historyStateBinding}_2,\text{initialAuthenticator})\)

\(\text{6.1. } \text{LL}_1\text{ObservedAuthenticators} = \{\text{initialAuthenticator}\}\)

\(\text{by } \langle 2\rangle 2 \text{ def } \text{initialAuthenticator}, \text{initialHistoryStateBinding}, \text{initialStateHash}, \text{initialPrivateStateEnc}\)

\(\text{6.2. } \text{LL}_1\text{HistoryStateBindingAuthenticated}(\text{historyStateBinding}_2)\)

\(\text{by } \langle 3\rangle 3\)

\(\text{6.3. QED}\)

\(\text{by } \langle 6\rangle 1, \langle 6\rangle 2 \text{ def } \text{LL}_1\text{HistoryStateBindingAuthenticated}\)

We also have to prove the appropriate types, which are the next four statements.
(5) 2. \( LL1NVRAM \text{.symmetricKey} \in \text{SymmetricKeyType} \)

(6) 1. \( LL1TypeInvariant \)

BY (2) 1

(6) 2. QED

BY (6) 1, LL1SubtypeImplicationLemmaDef LL1SubtypeImplication

(5) 3. \( \text{symmetricKey} \in \text{SymmetricKeyType} \)

BY (2) 2

(5) 4. \( \text{historyStateBinding2} \in \text{HashType} \)

BY (3) 2

(5) 5. \( \text{initialHistoryStateBinding} \in \text{HashType} \)

BY (2) 3

Now, we can apply the \( MACCollisionResistant \) property, which establishes that history state binding 2 equals the initial history state binding.

(5) 6. QED

BY (5) 1, (5) 2, (5) 3, (5) 4, (5) 5, MACCollisionResistant

DEF initialAuthenticator

Because each history state binding is equal to the initial state binding, the two history state bindings must equal each other.

(4) 3. QED

BY (4) 1, (4) 2

Since the two state bindings are equal, it follows from the collision resistance of the hash function that the two state hashes are equal.

(3) 5. QED

(4) 1. \( \text{stateHash1} \in \text{HashDomain} \)

(5) 1. \( \text{stateHash1} \in \text{HashType} \)

BY (3) 1

(5) 2. QED

BY (2) 1 DEF HashDomain

(4) 2. \( \text{stateHash2} \in \text{HashDomain} \)

(5) 1. \( \text{stateHash2} \in \text{HashType} \)

BY (3) 1

(5) 2. QED

BY (2) 1 DEF HashDomain

(4) 3. \( LL1NVRAM \text{.historySummary} \in \text{HashDomain} \)

(5) 1. \( LL1NVRAM \text{.historySummary} \in \text{HashType} \)

(6) 1. \( LL1TypeInvariant \)

BY (2) 1

(6) 2. QED

BY (6) 1, LL1SubtypeImplicationLemmaDef LL1SubtypeImplication

(5) 2. QED

BY (5) 1 DEF HashDomain

(4) 4. QED

BY (3) 4, (4) 1, (4) 2, (4) 3, HashCollisionResistant

DEF historyStateBinding1, historyStateBinding2

(2) 7. QED

BY (2) 4, (2) 5, (2) 6 DEF CorrectnessInvariants

For the induction step, we will need the type invariant and the unforgeablity invariant to be true in both the unprimed and primed states.

(1) 4. ( \( \land \) InclusionInvariant

\( \land \) CardinalityInvariant

\( \land \) UniquenessInvariant

\( \land \) \( LL1Next \)

LL1Vars

125
∧ LL1TypeInvariant
∧ LL1TypeInvariant'
∧ UnforgeabilityInvariant
∧ UnforgeabilityInvariant')
⇒
InclusionInvariant' ∧ CardinalityInvariant' ∧ UniquenessInvariant'

First, we assume the antecedents.

(2) 1. have ∧ InclusionInvariant
∧ CardinalityInvariant
∧ UniquenessInvariant
∧ [LL1Next]LL1Vars
∧ LL1TypeInvariant
∧ LL1TypeInvariant'
∧ UnforgeabilityInvariant
∧ UnforgeabilityInvariant'

The induction step includes two cases: stuttering and LL1Next actions. The stuttering case is a straightforward application of the InclusionUnchangedLemma, the CardinalityUnchangedLemma, and the UniquenessUnchangedLemma.

(2) 2. case unchanged LL1Vars
(3) 1. unchanged ⟨LL1NVRAM, LL1ObservedOutputs⟩
by (2) 2 def LL1Vars
(3) 2. ∀ historyStateBinding ∈ HashType :
unchanged LL1HistoryStateBindingAuthenticated(historyStateBinding)
(4) 1. unchanged ⟨LL1NVRAM, LL1ObservedAuthenticators⟩
by (2) 2 def LL1Vars
(4) 2. Qed
by (4) 1, UnchangedAuthenticatedHistoryStateBindingsLemma

(3) 3. InclusionInvariant'
(4) 1. InclusionInvariant ∧ LL1TypeInvariant ∧ LL1TypeInvariant'
by (2) 1
(4) 2. Qed
by (3) 1, (3) 2, (4) 1, InclusionUnchangedLemma

(3) 4. CardinalityInvariant'
(4) 1. CardinalityInvariant ∧ LL1TypeInvariant
by (2) 1
(4) 2. Qed
by (3) 1, (3) 2, (4) 1, CardinalityUnchangedLemma

(3) 5. UniquenessInvariant'
(4) 1. UniquenessInvariant ∧ LL1TypeInvariant
by (2) 1
(4) 2. Qed
by (3) 1, (3) 2, (4) 1, UniquenessUnchangedLemma

(3) 6. Qed
by (3) 3, (3) 4, (3) 5

We break down the LL1Next case into separate cases for each action.

(2) 3. case LL1Next
The LL1MakeInputAvailable case is a straightforward application of the InclusionUnchangedLemma, the CardinalityUnchangedLemma, and the UniquenessUnchangedLemma.

(3) 1. case LL1MakeInputAvailable
(4) 1. unchanged ⟨LL1NVRAM, LL1ObservedOutputs⟩
by (3) 1 def LL1MakeInputAvailable
The LL1PerformOperation case is the vast majority of this proof.

(3) CASE LL1PerformOperation

We pick some input for which LL1PerformOperation is true.

(4) 1. PICK input1 ∈ LL1AvailableInputs : LL1PerformOperation!(input1)!1

    BY (3) 2 DEF LL1PerformOperation

To simplify the writing of the proof, we re-state the definitions from the LL1PerformOperation action.

(4) 2. ∀ historyStateBinding ∈ HashType :

    UNCHANGED LL1HistoryStateBindingAuthenticated(historyStateBinding)

(4) 3. InclusionInvariant'

    (5) 1. InclusionInvariant ∧ LL1TypeInvariant ∧ LL1TypeInvariant'

    BY (2) 1

    2. QED

    BY (4) 1, (4) 2, (5) 1, InclusionUnchangedLemma

(4) 4. CardinalityInvariant'

    (5) 1. CardinalityInvariant ∧ LL1TypeInvariant

    BY (2) 1

    2. QED

    BY (4) 1, (4) 2, (5) 1, CardinalityUnchangedLemma

(4) 5. UniquenessInvariant'

    (5) 1. UniquenessInvariant ∧ LL1TypeInvariant

    BY (2) 1

    2. QED

    BY (4) 1, (4) 2, (5) 1, UniquenessUnchangedLemma

(4) 6. QED

    BY (4) 3, (4) 4, (4) 5

The LL1PerformOperation case is the vast majority of this proof.

(3) 2. CASE LL1PerformOperation

We pick some input for which LL1PerformOperation is true.

(4) 1. PICK input1 ∈ LL1AvailableInputs : LL1PerformOperation!(input1)!1

    BY (3) 2 DEF LL1PerformOperation

To simplify the writing of the proof, we re-state the definitions from the LL1PerformOperation action.

(4) 2. ∀ historyStateBinding ∈ HashType :

    UNCHANGED LL1HistoryStateBindingAuthenticated(historyStateBinding)

(4) 3. InclusionInvariant'

    (5) 1. InclusionInvariant ∧ LL1TypeInvariant ∧ LL1TypeInvariant'

    BY (2) 1

    2. QED

    BY (4) 1, (4) 2, (5) 1, InclusionUnchangedLemma

(4) 4. CardinalityInvariant'

    (5) 1. CardinalityInvariant ∧ LL1TypeInvariant

    BY (2) 1

    2. QED

    BY (4) 1, (4) 2, (5) 1, CardinalityUnchangedLemma

(4) 5. UniquenessInvariant'

    (5) 1. UniquenessInvariant ∧ LL1TypeInvariant

    BY (2) 1

    2. QED

    BY (4) 1, (4) 2, (5) 1, UniquenessUnchangedLemma

(4) 6. QED

    BY (4) 3, (4) 4, (4) 5

The LL1PerformOperation case is the vast majority of this proof.
\[ newAuthenticator \in MACType \]

(5.1) \textit{input1} \in LL1AvailableInputs

BY (4.1)

(5.2) \textit{LL1TypeInvariant}

BY (2.1)

(5.3) \textit{QED}

BY (5.1), (5.2), \textit{LL1PerformOperationDefsTypeSafeLemma}

We hide the definitions, so they don’t overwhelm the prover. We’ll pull them in as necessary below.

(4) \textit{HIDE DEF stateHash, historyStateBinding, privateState, sResult, newPrivateStateEnc, newHistorySummary, newStateHash, newHistoryStateBinding, newAuthenticator}

We prove some simple type statements up front, to avoid needing to repeat these multiple times below.

(4.3) \& LL1NVRAM.historySummary \in HashType

\& LL1NVRAM.symmetricKey \in SymmetricKeyType

(5.1) \textit{LL1TypeInvariant}

BY (2.1)

(5.2) \textit{QED}

BY (5.1), \textit{LL1SubtypeImplicationLemmaDef LL1SubtypeImplication}

(4.4) \& LL1NVRAM.historySummary’ \in HashType

\& LL1NVRAM.symmetricKey’ \in SymmetricKeyType

(5.1) \textit{LL1TypeInvariant’}

BY (2.1)

(5.2) \textit{QED}

BY (5.1), \textit{LL1SubtypeImplicationLemmaDef LL1SubtypeImplication}

We also prove that \textit{LL1NVRAMHistorySummaryUncorrupted} predicate is true. This follows from the enablement conditions for \textit{LL1PerformOperation}.

(4.5) \textit{LL1NVRAMHistorySummaryUncorrupted}

We prove this directly from the definition of the \textit{LL1NVRAMHistorySummaryUncorrupted} predicate. There is one type assumption and one antecedent in the implication.

(5.1) \textit{stateHash} \in HashType

BY (4.2)

(5.2) \textit{LL1HistoryStateBindingAuthenticated(historyStateBinding)}

The authenticator in the RAM is a witness, because \textit{TMPPerformOperation} ensures that it validates the history state binding.

(6.1) \textit{ValidateMAC}(

\begin{align*}
& LL1NVRAM.symmetricKey, \text{historyStateBinding}, LL1RAM.authenticator \\
\end{align*}

) \textit{BY (4.1) DEF historyStateBinding, stateHash}

The \textit{UnforgeabilityInvariant} then ensures that this authenticator is in the set of observed authenticators.

(6.2) \textit{LL1RAM.authenticator} \in LL1ObservedAuthenticators

(7.1) \textit{UnforgeabilityInvariant}

BY (2.1)

(7.2) \textit{historyStateBinding} \in HashType

BY (4.2)

(7.3) \textit{QED}

BY (6.1), (7.1), (7.2) DEF \textit{UnforgeabilityInvariant}, \textit{LL1HistoryStateBindingAuthenticated}

These two conditions satisfy the definition of \textit{LL1HistoryStateBindingAuthenticated}.

(6.3) \textit{QED}

BY (6.1), (6.2) DEF \textit{LL1HistoryStateBindingAuthenticated}

(5.3) \textit{LL1NVRAM.historySummary} = LL1RAM.historySummary

BY (4.1)

(5.4) \textit{QED}

BY (5.1), (5.2), (5.3) DEF \textit{LL1NVRAMHistorySummaryUncorrupted, historyStateBinding}
We’ll prove each of the three invariants separately. First, we’ll prove that the \textit{InclusionInvariant} holds in the primed state.

(4)6. \textit{InclusionInvariant’}

To prove the universally quantified expression, we take a new set of variables of the appropriate types. For the \textsc{take} step to be meaningful to the prover, first we have to tell the prover to expand the definition of \textit{InclusionInvariant}, so it will see the universally quantified expression therein.

(5) \textsc{use def} \textit{InclusionInvariant}

(5) \textsc{take input2 \in InputType,}
\textsc{historySummary} \in \text{HashType},
\text{publicState} \in \text{PublicStateType},
\text{privateStateEnc} \in \text{PrivateStateEncType}

To simplify the writing of the proof, we re-state the definitions from the \textit{InclusionInvariant}.

(5) inclStateHash \triangleq \text{Hash(publicState, privateStateEnc)}
(5) inclHistoryStateBinding \triangleq \text{Hash(historySummary, inclStateHash)}
(5) inclPrivateState \triangleq \text{SymmetricDecrypt(LL1NVRAM.symmetricKey, privateStateEnc)}
(5) inclSResult \triangleq \text{Service(publicState, inclPrivateState, input2)}
(5) inclNewPrivateStateEnc \triangleq \text{SymmetricEncrypt(LL1NVRAM.symmetricKey, inclSResult.newPrivateState)}
(5) inclNewStateHash \triangleq \text{Hash(inclSResult.newPublicState, inclNewPrivateStateEnc)}
(5) inclNewHistoryStateBinding \triangleq \text{Hash(LL1NVRAM.historySummary, inclNewStateHash)}

We then assert the type safety of these definitions, with the help of the \textit{InclusionInvariantDefsTypeSafeLemma}.

(5)2. \wedge \text{inclStateHash} \in \text{HashType}
\wedge \text{inclHistoryStateBinding} \in \text{HashType}
\wedge \text{inclPrivateState} \in \text{PrivateStateType}
\wedge \text{inclSResult} \in \text{ServiceResultType}
\wedge \text{inclSResult.newPublicState} \in \text{PublicStateType}
\wedge \text{inclSResult.newPrivateState} \in \text{PrivateStateType}
\wedge \text{inclSResult.output} \in \text{OutputType}
\wedge \text{inclNewPrivateStateEnc} \in \text{PrivateStateEncType}
\wedge \text{inclNewStateHash} \in \text{HashType}
\wedge \text{inclNewHistoryStateBinding} \in \text{HashType}

(6)1. LL1TypeInvariant
\textsc{by} (2)1

(6)2. \textsc{qed}
\textsc{by} (5)1, (6)1, \textit{InclusionInvariantDefsTypeSafeLemma}

The \textit{InclusionInvariant} states an implication. To prove this, it suffices to assume the antecedent and prove the consequent.

(5)3. \textsc{suffices}
\textsc{assume}
\wedge \text{LL1NVRAM.historySummary'} = \text{Hash(historySummary, input2)}
\wedge \text{LL1HistoryStateBindingAuthenticated(inclHistoryStateBinding)'}
\textsc{prove}
\wedge \text{inclSResult.output'} \in \text{LL1ObservedOutputs'}
\wedge \text{LL1HistoryStateBindingAuthenticated(inclNewHistoryStateBinding)'}
\textsc{obvious}

We hide the definition of \textit{InclusionInvariant} and the definitions from the \textit{InclusionInvariant}.

(5) \textsc{hide def} \textit{InclusionInvariant}

(5) \textsc{hide def inclStateHash, inclHistoryStateBinding, inclPrivateState, inclSResult, inclNewPrivateStateEnc, inclNewStateHash, inclNewHistoryStateBinding}

Starting the \textit{InclusionInvariant} proof proper, there are four main steps.
First, we prove that the history summary in the NVRAM matches the history summary that satisfies the antecedent condition, and that the input supplied to the $\text{LL1PerformOperation}$ action matches the input that satisfies the antecedent condition.

Second, we prove that the public and private state in the RAM matches the public and private state that satisfies the antecedent condition.

The third and fourth steps each use the above results to prove one of the conjuncts in the consequent of the $\text{InclusionInvariant}$.  

(5)4.  $\wedge \text{LL1NVRAM.historySummary} = \text{historySummary}$  
$\wedge \text{input1} = \text{input2}$  

To prove the above equivalences, we will use the $\text{HashCollisionResistant}$ property. To use this, we first have to prove some simple types.  

(6)1. $\text{LL1NVRAM.historySummary} \in \text{HashDomain}$  
(7)1. $\text{LL1NVRAM.historySummary} \in \text{HashType}$  

BY ⟨4⟩3  
(7)2. QED  
BY ⟨7⟩1 DEF HashDomain  

(6)2. $\text{input1} \in \text{HashDomain}$  
(7)1. $\text{input1} \in \text{InputType}$  

⟨8⟩1. $\text{input1} \in \text{LL1AvailableInputs}$  
BY ⟨4⟩1  
⟨8⟩2. $\text{LL1AvailableInputs} \subseteq \text{InputType}$  
(9)1. $\text{LL1TypeInvariant}$  
BY ⟨2⟩1  
(9)2. QED  
BY ⟨9⟩1 DEF LL1TypeInvariant  
⟨8⟩3. QED  
BY ⟨8⟩1, ⟨8⟩2  
(7)2. QED  
BY ⟨7⟩1 DEF HashDomain  

(6)3. $\text{historySummary} \in \text{HashDomain}$  
(7)1. $\text{historySummary} \in \text{HashType}$  
BY ⟨5⟩1  
(7)2. QED  
BY ⟨7⟩1 DEF HashDomain  

(6)4. $\text{input2} \in \text{HashDomain}$  
(7)1. $\text{input2} \in \text{InputType}$  
BY ⟨5⟩1  
(7)2. QED  
BY ⟨7⟩1 DEF HashDomain  

Then, we have to prove that the hashes are equal, which follows from the definition of the new history summary produced by $\text{LL1PerformOperation}$.  

(6)5. $\text{Hash(}\text{LL1NVRAM.historySummary}, \text{input1}) = \text{Hash(}\text{historySummary}, \text{input2})$  
(7)1. $\text{newHistorySummary} = \text{Hash(}\text{LL1NVRAM.historySummary}, \text{input1})$  
BY DEF newHistorySummary  
(7)2. $\text{LL1NVRAM.historySummary}' = \text{Hash(}\text{historySummary}, \text{input2})$  
BY ⟨5⟩3  
(7)3. $\text{LL1NVRAM.historySummary}' = \text{newHistorySummary}$  
⟨8⟩1. $\text{LL1NVRAM'} = \{\text{historySummary} \mapsto \text{newHistorySummary}, \text{symmetricKey} \mapsto \text{LL1NVRAM.symmetricKey}\}$  
BY ⟨4⟩1 DEF newHistorySummary, newPrivateStateEnc, sResult, privateState  
⟨8⟩2. QED
Ideally, this QED step should just read:

By (6)1, (6)2, (6)3, (6)4, (6)5, HashCollisionResistant

However, the prover seems to get a little confused in this instance. We make life easier for the prover by defining some local variables and hiding their definitions before appealing to the HashCollisionResistant assumption.

(7) 1. $h_1 a \triangleq LL1NVRAM.historySummary$
(7) 2. $h_2 a \triangleq input1$
(7) 3. $h_1 b \triangleq historySummary$
(7) 4. $h_2 b \triangleq input2$

The second main step in the InclusionInvariant proof is to prove that the public and private state in the RAM matches the public and private state that satisfies the antecedent condition.

Most of the work of proving this step is proving that the public state in the RAM matches the public state that satisfies the antecedent condition, and that the encrypted private state in the RAM matches the encrypted private state that satisfies the antecedent condition.

To prove the above equivalences, we will use the HashCollisionResistant property. To use this, we first have to prove some simple types.
2. \text{QED}
\begin{align*}
&\text{by (9) 1, } LL1\text{SubtypeImplicationLemma} \\
&\text{def } LL1\text{SubtypeImplication}
\end{align*}
\begin{align*}
\langle 8 \rangle 2. \text{QED} \\
&\text{by (8) 1 def } HashDomain
\end{align*}
\begin{align*}
\langle 7 \rangle 3. \text{privateStateEnc } \in HashDomain \\
&\text{by (5) 1} \\
\langle 8 \rangle 2. \text{QED} \\
&\text{by (8) 1 def } HashDomain
\end{align*}

Then, we'll show that the state hash defined in the InclusionInvariant matches the state hash defined in LL1\text{PerformOperation}. This is the hash equivalence we will use in our appeal to the HashCollisionResistant property.

This next step is quite involved, so the remainder of this proof follows much further down.

\begin{align*}
\langle 7 \rangle 4. \text{inclStateHash'} = stateHash \\
&\text{We can show that the state hashes match using the UniquenessInvariant. This will require proving some simple type statements, immediately below, and then proving that the two history state bindings corresponding to the two hashes are both authenticated.}
\end{align*}
\begin{align*}
\langle 8 \rangle 1. \text{inclStateHash'} \in HashType \\
&\text{by (4) 2}
\end{align*}
The first history state binding we have to prove to be authenticated is the history state binding defined in the InclusionInvariant. This is by far the more involved of the two.

Our strategy is to first prove that this history state binding is authenticated in the primed state. Then, we prove that this history state binding is not validated by the new authenticator defined in the LL1\text{PerformOperation} action. Since this new authenticator is the only element that is in the primed set of observed authenticators but not in the unprimed set of observed authenticators, it follows that the history state binding is authenticated by an authenticator in the unprimed set of authenticators, so it is authenticated in the unprimed state.

\begin{align*}
\langle 8 \rangle 3. \text{LL1HistoryStateBindingAuthenticated(inclHistoryStateBinding)} \\
&\text{The history state binding defined by the InclusionInvariant is authenticated in the primed state by hypothesis.}
\end{align*}
\begin{align*}
\langle 9 \rangle 1. \text{LL1HistoryStateBindingAuthenticated(inclHistoryStateBinding)}' \\
&\text{by (5) 3}
\end{align*}
We have to prove that the history state binding defined by the InclusionInvariant is not authenticated by the new authenticator defined in the LL1\text{PerformOperation} action.

\begin{align*}
\langle 9 \rangle 2. \neg \text{ValidateMAC(} \\
&\text{LL1NVRAM.symmetricKey,} \\
&\text{inclHistoryStateBinding,} \\
&\text{newAuthenticator)}
\end{align*}
Our strategy is to use proof by contradiction. First, we will show that if the history state binding defined by the InclusionInvariant were authenticated by the new authenticator defined in the LL1PerformOperation action, then this history state binding would equal the new history state binding defined in the LL1PerformOperation action. This then leads to the conclusion that the two history summaries in each of these history state bindings are equal. However, we will show that these two history summaries cannot be equal, because one was generated from a hash of the other.

\langle 10 \rangle 1. \text{SUFFICES}
\begin{align*}
\text{ASSUME} & \quad \text{ValidateMAC}( \\
& \quad \text{LL1NVRAM.symmetricKey,} \\
& \quad \text{inclHistoryStateBinding,} \\
& \quad \text{newAuthenticator}) \\
\text{PROVE} & \quad \text{FALSE} \\
\end{align*}

\text{OBSVIOUS}

In our contradictory universe, the history state binding defined by the InclusionInvariant equals the new history state binding defined in the LL1PerformOperation action, due to the collision resistance of the MAC. We merely need to prove some types, and then we can employ the MACCollisionResistant property directly.

\langle 10 \rangle 2. \text{inclHistoryStateBinding} = \text{newHistoryStateBinding}
\langle 11 \rangle 1. \text{LL1NVRAM.symmetricKey} \in \text{SymmetricKeyType}
\text{BY} \ (4) \ 3
\langle 11 \rangle 2. \text{inclHistoryStateBinding} \in \text{HashType}
\text{BY} \ (5) \ 2
\langle 11 \rangle 3. \text{newHistoryStateBinding} \in \text{HashType}
\text{BY} \ (4) \ 2
\langle 11 \rangle 4. \text{QED}
\text{BY} \ (10)1, (11)1, (11)2, (11)3, \text{MACCollisionResistant}
\text{DEF} \ \text{newAuthenticator}

In our contradictory universe, the history summary quantified by the InclusionInvariant equals the new history summary defined in the LL1PerformOperation action, due to the collision resistance of the hash function. We merely need to prove some types, and then we can employ the HashCollisionResistant property directly.

\langle 10 \rangle 3. \text{historySummary} = \text{newHistorySummary}
\langle 11 \rangle 1. \text{historySummary} \in \text{HashDomain}
\langle 12 \rangle 1. \text{historySummary} \in \text{HashType}
\text{BY} \ (5) \ 3
\langle 12 \rangle 2. \text{QED}
\text{BY} \ (12)1 \text{ DEF HashDomain}
\langle 11 \rangle 2. \text{inclStateHash} \in \text{HashDomain}
\langle 12 \rangle 1. \text{inclStateHash} \in \text{HashType}
\text{BY} \ (5) \ 2
\langle 12 \rangle 2. \text{QED}
\text{BY} \ (12)1 \text{ DEF HashDomain}
\langle 11 \rangle 3. \text{newHistorySummary} \in \text{HashDomain}
\langle 12 \rangle 1. \text{newHistorySummary} \in \text{HashType}
\text{BY} \ (4) \ 2
\langle 12 \rangle 2. \text{QED}
\text{BY} \ (12)1 \text{ DEF HashDomain}
\langle 11 \rangle 4. \text{newStateHash} \in \text{HashDomain}
\langle 12 \rangle 1. \text{newStateHash} \in \text{HashType}
\text{BY} \ (4) \ 2
\langle 12 \rangle 2. \text{QED}
\text{BY} \ (12)1 \text{ DEF HashDomain}
Back in the real universe, we will prove that the history summary quantified by the InclusionInvariant cannot equal the new history summary defined in the LL1PerformOperation action. The proof relies on the property of hash cardinality, though not on the CardinalityInvariant. Basically, we prove that the cardinality of the history summary in the new authenticator defined by LL1PerformOperation is one greater than the hash cardinality of the history summary previously in the NVRAM. Since the hash cardinalities differ, it follows that the history summaries differ. The proof is tedious but straightforward.

\( \text{historySummary} \neq \text{newHistorySummary} \)

First, we prove a bunch of types that are needed by the hash cardinality assumptions or for proving basic arithmetic.

\( \text{input1} \in \text{InputType} \)

\( \text{input1} \in \text{LL1AvailableInputs} \)

\( \text{LL1AvailableInputs} \subseteq \text{InputType} \)

\( \text{LL1TypeInvariant} \)

\( \text{newHistorySummary} \in \text{HashDomain} \)

\( \text{newHistorySummary} \in \text{HashType} \)

\( \text{HashCardinality} (\text{newHistorySummary}) \in \text{Nat} \)

With the type statements out of the way, we can construct a simple inequality. We do this in four linear steps.

\( \text{HashCardinality} (\text{newHistorySummary}) = \text{HashCardinality} (\text{LL1NVRAM.historySummary}) + \text{HashCardinality} (\text{input1}) + 1 \)
The second history state binding we have to prove to be authenticated is the history state binding defined \[
\langle 8 \rangle.\quad \text{HashCardinality}(\text{newHistorySummary}) = \text{HashCardinality}(\text{historySummary}) + 1
\]
\[
\langle 12 \rangle.1.\quad \text{LL1NVRAM.historySummary} = \text{historySummary}
\]
\[
\text{by (5)4}
\]
\[
\langle 12 \rangle.2.\quad \text{QED}
\]
\[
\text{by (11)7, (12)1}
\]
\[
\langle 11 \rangle.9.\quad \text{HashCardinality}(\text{newHistorySummary}) = \text{HashCardinality}(\text{historySummary}) + 1
\]
\[
\langle 12 \rangle.1.\quad \text{HashCardinality}(\text{input1}) = 0
\]
\[
\text{by (11)1, InputCardinalityZero}
\]
\[
\langle 12 \rangle.2.\quad \text{QED}
\]
\[
\text{by (11)5, (11)6, (11)8, (12)1}
\]
\[
\langle 11 \rangle.10.\quad \text{HashCardinality}(\text{newHistorySummary}) \neq \text{HashCardinality}(\text{historySummary})
\]
\[
\text{by (11)5, (11)6, (11)9}
\]
Since the hash cardinalities differ, it follows that the history summaries differ.
\[
\langle 11 \rangle.11.\quad \text{QED}
\]
\[
\text{by (11)10}
\]
The required contradiction follows from the previous two steps.
\[
\langle 10 \rangle.5.\quad \text{QED}
\]
\[
\text{by (10)3, (10)4}
\]
Before completing the proof, we need to establish that the symmetric key in the NVRAM has not changed, because the \text{LL1HistoryStateBindingAuthenticated} predicate implicitly refers to this variable.
\[
\langle 9 \rangle.3.\quad \text{UNCHANGED LL1NVRAM.symmetricKey}
\]
\[
\text{by (2)1, SymmetricKeyConstantLemma}
\]
We also need to establish that the new authenticator defined by the \text{LL1PerformOperation} action is the only element that is in the primed set of observed authenticators but not in the unprimed set of observed authenticators.
\[
\langle 9 \rangle.4.\quad \text{LL1ObservedAuthenticators'} = \text{LL1ObservedAuthenticators} \cup \{\text{newAuthenticator}\}
\]
\[
\text{by (4)1 def newAuthenticator, newHistoryStateBinding, newStateHash, newHistorySummary, newPrivateStateEnc, sResult, privateState}
\]
The conclusion follows directly.
\[
\langle 9 \rangle.5.\quad \text{QED}
\]
\[
\text{by (9)1, (9)2, (9)3, (9)4 def LL1HistoryStateBindingAuthenticated}
\]
The second history state binding we have to prove to be authenticated is the history state binding defined in the \text{LL1PerformOperation} action. We need to show that it is authenticated by some authenticator in the set of observed authenticators.
\[
\langle 8 \rangle.4.\quad \text{LL1HistoryStateBindingAuthenticated(historyStateBinding)}
\]
The authenticator in the RAM is a witness, because \text{TMPPerformOperation} ensures that it validates the history state binding.
\[
\langle 9 \rangle.1.\quad \text{ValidateMAC(\text{LL1NVRAM.symmetricKey, historyStateBinding, LL1RAM.authenticator})}
\]
\[
\text{by (4)1 def historyStateBinding, stateHash}
\]
The \text{UnforgeabilityInvariant} then ensures that this authenticator is in the set of observed authenticators.
\[
\langle 9 \rangle.2.\quad \text{LL1RAM.authenticator} \in \text{LL1ObservedAuthenticators}
\]
\[
\text{by (2)1}
\]
\[
\langle 10 \rangle.2.\quad \text{historyStateBinding} \in \text{HashType}
\]
\[
\text{by (4)2}
\]
\[
\langle 10 \rangle.3.\quad \text{QED}
\]
\[
\text{by (9)1, (10)1, (10)2 def UnforgeabilityInvariant, LL1HistoryStateBindingAuthenticated}
\]
These two conditions satisfy the definition of \text{LL1HistoryStateBindingAuthenticated}.
The conclusion follows fairly directly from the UniquenessInvariant. However, one small hitch is that the UniquenessInvariant requires both history state bindings to be defined using LL1NVRAM.historySummary, but inclHistoryStateBinding is defined using historySummary rather than LL1NVRAM.historySummary. Therefore, we have to add in the fact that LL1NVRAM.historySummary = historySummary.

The conclusion follows directly from applying the HashCollisionResistant property.

The remainder of this step follows trivially.

This follows fairly directly from the just-proven facts that the variables that satisfy the antecedent conditions match the corresponding values in the NVRAM. This follows fairly directly from the just-proven facts that the variables that satisfy the antecedent conditions match the corresponding values in the NVRAM. Therefore, we have to add in the fact that LL1NVRAM.historySummary = historySummary.

The third main step in the InclusionInvariant proof is to prove the first conjunct in the consequent of the InclusionInvariant, namely that the primed output of the service is in the primed set of observed outputs.
The fourth main step in the InclusionInvariant proof is to prove the second conjunct in the consequent of the InclusionInvariant, namely that the new history state binding defined in the InclusionInvariant is authenticated in the primed state.

We need to show that there exists some authenticator in the primed set of observed authenticators that authenticates this history state binding. Our witness is the new authenticator defined by the LL1PerformOperation action.

(5)7. LL1HistoryStateBindingAuthenticated(inclNewHistoryStateBinding)'

The new authenticator defined in the LL1PerformOperation action is unioned into the set of observed authenticators by the LL1PerformOperation action.

(6)1. newAuthenticator ∈ LL1ObservedAuthenticators'

(7)1. LL1ObservedAuthenticators' =

LL1ObservedAuthenticators ∪ {newAuthenticator}

BY (4)1 DEF newAuthenticator, newHistoryStateBinding, newStateHash, newHistorySummary, newPrivateStateEnc, sResult, privateState

(7)2. QED
BY (7)1

To prove that the new authenticator defined by the LL1PerformOperation action authenticates the new history state binding defined in the InclusionInvariant, we show that this authenticator was generated using this history state binding and using the same key.

(6)2. ValidateMAC(LL1NVRAM.symmetricKey', inclNewHistoryStateBinding', newAuthenticator)

First, we show that the new history state binding defined by the LL1PerformOperation action is equal to the primed state of the new history state binding defined in the InclusionInvariant.

(7)1. inclNewHistoryStateBinding' = newHistoryStateBinding

The proof is fairly straightforward. Using the above-proven facts that the variables that satisfy the antecedent conditions match the corresponding values in the NVRAM and the input value provided to the LL1PerformOperation action, we show that the results of the service in the primed state of the InclusionInvariant are the same as the results of the service in the LL1PerformOperation action. From there, we show that the state hashes are equal and that the history summaries are equal, which together imply that the state bindings are equal.

⟨8⟩1. UNCHANGED LL1NVRAM.symmetricKey
BY (2)1, SymmetricKeyConstantLemma
⟨8⟩2. inclSResult' = sResult

⟨9⟩1. ∧ publicState = LL1RAM.publicState
∧ inclPrivateState' = privateState

BY (5)5
⟨9⟩2. input2 = input1
BY (5)4
⟨9⟩3. QED
BY (9)1, (9)2 DEF inclSResult, sResult
 ⟨8⟩3. inclSResult'.newPublicState = sResult.newPublicState
BY (8)2
 ⟨8⟩4. inclSResult'.newPrivateState = sResult.newPrivateState
BY (8)2
 ⟨8⟩5. inclNewPrivateStateEnc' = newPrivateStateEnc
BY (8)1, (8)4 DEF inclNewPrivateStateEnc, newPrivateStateEnc
 ⟨8⟩6. inclNewStateHash' = newStateHash
BY (8)3, (8)5 DEF inclNewStateHash, newStateHash
Given that the state bindings are equal (and showing that the symmetric keys are equal, we can show that the new authenticator defined by the \textit{LL1PerformOperation} action is equal to a MAC generated with the same inputs we are attempting to validate.

\[ \text{newAuthenticator} = \text{GenerateMAC}(\text{LL1NVRAM}.\text{symmetricKey}', \text{inclNewHistoryStateBinding}') \]

We can thus use the \textit{MACComplete} property to show that the generated MAC validates appropriately. To do this, we first need to prove some types.

Since we have no lemma to prove the following type, we include its entire type proof here.

1. LL1NVRAM'. \text{historySummary}' = newHistorySummary
2. LL1NVRAM' = [\text{historySummary} \rightarrow newHistorySummary, 
\text{symmetricKey} \rightarrow LL1NVRAM.\text{symmetricKey}]
3. by (4) 1 DEF newHistorySummary, newPrivateStateEnc, sResult, privateState
4. QED by (10) 1
5. QED by (8) 6, (8) 7 DEF inclNewHistoryStateBinding, newHistoryStateBinding
6. \text{inclNewStateHash}' \in HashDomain
7. LL1NVRAM.\text{historySummary}' \in HashType
8. by (9) 1 DEF HashDomain
9. QED by (9) 1, (9) 2, (9) 3, ServiceTypeSafe DEF inclSResult
10. LL1NVRAM.\text{historySummary}' \in HashDomain
11. LL1NVRAM.\text{historySummary}' \in HashType
12. by (9) 1, LL1SubtypeImplicationLemma DEF LL1SubtypeImplication
13. QED by (9) 1, (9) 2, LL1SubtypeImplicationLemma DEF inclPrivateStateEnc
14. inclSResult' \in ServiceResultType
15. by (5) 1
16. QED by (9) 2, inclSResult' \in ServiceResultType
17. inclPrivateState' \in PrivateStateType
18. by (11) 1, LL1SubtypeImplicationLemma DEF LL1SubtypeImplication
19. QED by (11) 2
20. inclPrivateState' \in PrivateStateType
21. by (11) 2
22. QED by (11) 1, LL1SubtypeImplicationLemma DEF LL1SubtypeImplication
Then, we appeal to the MACComplete property in a straightforward way.

Second, we'll prove the CardinalityInvariant in the primed state.

To prove the universally quantified expression, we take a new set of variables of the appropriate types. For the take step to be meaningful to the prover, first we have to tell the prover to expand the definition of CardinalityInvariant, so it will see the universally quantified expression therein.

We then assert the type safety of these definitions, with the help of the CardinalityInvariantDefsTypeSafeLemma.

The CardinalityInvariant states an implication. To prove this, it suffices to assume the antecedent and prove the consequent.

Assume

\[ \text{LL1NVRAMHistorySummaryUncorrupted}' \]
\[ \text{ LL1HistoryStateBindingAuthenticated}(\text{cardHistoryStateBinding}') \]
We then hide the definitions.

First, we'll prove that the hash cardinality of the history summary in the NVRAM does not decrease when a \emph{LL1PerformOperation} action occurs. This follows from the fact that the hash cardinality of the history summary in the LL1NVRAM increases by one when a \emph{LL1PerformOperation} action occurs. This is straightforward though somewhat tedious.

(5)4. \texttt{HashCardinality(\texttt{LL1NVRAM.historySummary}) ≤ HashCardinality(\texttt{LL1NVRAM.historySummary}')}

First, we prove a bunch of types that are needed by the hash cardinality assumptions or for proving basic arithmetic.

(6)1. \texttt{input1 ∈ InputType}
(7)1. \texttt{input1 ∈ LL1AvailableInputs}
   \hspace{1em} \texttt{BY (4)1}
(7)2. \texttt{LL1AvailableInputs ⊆ InputType}
   \hspace{1em} \texttt{BY (2)1}
   \hspace{1em} \texttt{\lll (8)1. LL1TypeInvariant}
   \hspace{1em} \texttt{BY (2)1}
   \hspace{1em} \texttt{\lll (8)2. QED}
   \hspace{1em} \texttt{\lll (7)3. QED}
   \hspace{1em} \texttt{\lll BY (7)1, (7)2}
(6)2. \texttt{LL1NVRAM.historySummary ∈ HashDomain}
(7)1. \texttt{LL1NVRAM.historySummary ∈ HashType}
   \hspace{1em} \texttt{BY (4)3}
(7)2. QED
   \hspace{1em} \texttt{\lll BY (7)1 \emph{HashDomain}}
(6)3. \texttt{LL1NVRAM.historySummary' ∈ HashDomain}
(7)1. \texttt{LL1NVRAM.historySummary' ∈ HashType}
   \hspace{1em} \texttt{BY (4)4}
(7)2. QED
   \hspace{1em} \texttt{\lll BY (7)1 \emph{HashDomain}}
(6)4. \texttt{input1 ∈ HashDomain}
   \hspace{1em} \texttt{\lll BY (6)1 \emph{HashDomain}}
(6)5. \texttt{HashCardinality(\texttt{LL1NVRAM.historySummary}) ∈ Nat}
   \hspace{1em} \texttt{\lll BY (6)2, \texttt{HashCardinalityTypeSafe\texttt{DEF HashDomain}}}
(6)6. \texttt{HashCardinality(\texttt{LL1NVRAM.historySummary'}) ∈ Nat}
   \hspace{1em} \texttt{\lll BY (6)3, \texttt{HashCardinalityTypeSafe\texttt{DEF HashDomain}}}

With the type statements out of the way, we can construct a simple inequality. We do this in four linear steps.

(6)7. \texttt{HashCardinality(\texttt{Hash(\texttt{LL1NVRAM.historySummary, input1})}) =}
   \hspace{2em} \texttt{HashCardinality(\texttt{LL1NVRAM.historySummary}) + HashCardinality(input1) + 1}
   \hspace{1em} \texttt{\lll BY (6)2, (6)4, \texttt{HashCardinalityAccumulative}}
(6)8. \texttt{HashCardinality(\texttt{LL1NVRAM.historySummary'}) =}
   \hspace{2em} \texttt{HashCardinality(\texttt{LL1NVRAM.historySummary}) + HashCardinality(input1) + 1}
   \hspace{1em} \texttt{\lll (7)1. LL1NVRAM.historySummary' = \texttt{Hash(\texttt{LL1NVRAM.historySummary, input1})}}
   \hspace{1em} \texttt{\lll (8)1. newHistorySummary = \texttt{Hash(\texttt{LL1NVRAM.historySummary, input1})}}
   \hspace{1em} \texttt{\lll BY \texttt{DEF newHistorySummary}}
   \hspace{1em} \texttt{\lll (8)2. LL1NVRAM.historySummary' = newHistorySummary}
   \hspace{1em} \texttt{\lll (9)1. LL1NVRAM' = [}
Next, we'll prove that either the history state binding is authenticated in the unprimed state or the history state binding is authenticated by the new authenticator defined by the \( LL_1 \) \text{PerformOperation} action. This follows from the fact that the primed set of authenticators is constructed as the union of the unprimed set and the new authenticator.

\[
(5) \quad \lor \quad LL_1 \text{HistoryStateBindingAuthenticated}(\text{cardHistoryStateBinding})
\]

\[
(6) \quad \land \quad \text{ValidateMAC}(LL_1 \text{NVRAM}.\text{symmetricKey}, \text{cardHistoryStateBinding}, \text{newAuthenticator})
\]

Given the above disjunction, we proceed via case analysis. First, we consider the case in which the history state binding is authenticated in the unprimed state.

In this case, because the \( \text{CardinalityInvariant} \) is true in the unprimed state, we can prove that the hash cardinality of the history summary that satisfies the antecedent is less than or equal to the hash cardinality of the history summary in the unprimed \( \text{NVRAM} \). Since the less-than-or-equal-to relation is transitive, this leads directly to our proof goal.
The consequent of the implication in the \textit{CardinalityInvariant} follows directly.

\[ x \triangleq \text{HashCardinality(historySummary)} \]
\[ y \triangleq \text{HashCardinality(LL1NVRAM.historySummary)} \]
\[ z \triangleq \text{HashCardinality(LL1NVRAM.historySummary')} \]

Ideally, this $\text{QED}$ step should just read:

\[ \text{by } (5)3, \ (6)2, \ (6)3, \ (6)4, \ (6)5, \ \text{LEQTransitive} \]

However, the prover seems to get a little confused in this instance. We make life easier for the prover by defining some local variables and hiding their definitions before appealing to the $\text{LEQTransitive}$ assumption.

\[ x \leq y \]
\[ y \leq z \]

\[ \text{by } (6)2 \]
\[ (7) \]
\[ \text{by } (5)4 \]
\[ (7) \]
\[ \text{by } (5)4 \]
\[ (7) \]
\[ \text{by } (6)5 \]
\[ (7) \]
\[ \text{by } (6)4 \]
\[ (7) \]
\[ \text{by } (6)5 \]
\[ (7) \]
\[ \text{by } (6)4 \]
\[ (7) \]
\[ \text{by } (5)4 \]
\[ (7) \]
\[ \text{by } (6)5 \]
In the other case, the history state binding is authenticated by the new authenticator defined by the LL1PerformOperation action.

(5) 7. CASE ValidateMAC(LL1NVRAM.symmetricKey, cardHistoryStateBinding, newAuthenticator)

Since the new authenticator authenticates the new history state binding defined in the LL1PerformOperation action, it follows that the history state bindings are equal, by the MACCollisionResistant property.

(6) 1. cardHistoryStateBinding = newHistoryStateBinding
(7) 1. LL1NVRAM.symmetricKey ∈ SymmetricKeyType
    BY (4) 3
(7) 3. cardHistoryStateBinding ∈ HashType
    BY (5) 2
(7) 4. newHistoryStateBinding ∈ HashType
    BY (4) 2
(7) 5. QED
    BY (5) 7, (7) 1, (7) 3, (7) 4, MACCollisionResistant
DEF newAuthenticator

Since the inputs-state bindings are equal, it follows that the history summaries are equal, by the HashCollisionResistant property and the fact that the history summary in the primed NVRAM equals the new history summary defined in the LL1PerformOperation action.

(6) 2. historySummary = LL1NVRAM.historySummary′
(7) 1. historySummary = newHistorySummary
    ⟨8⟩ 1. historySummary ∈ HashDomain
        ⟨9⟩ 1. historySummary ∈ HashType
            BY (5) 5
        ⟨9⟩ 2. QED
            BY (9) 1 DEF HashDomain
    ⟨8⟩ 2. stateHash2 ∈ HashDomain
        ⟨9⟩ 1. stateHash2 ∈ HashType
            BY (5) 1
        ⟨9⟩ 2. QED
            BY (9) 1 DEF HashDomain
    ⟨8⟩ 3. newHistorySummary ∈ HashDomain
        ⟨9⟩ 1. newHistorySummary ∈ HashType
            BY (4) 2
        ⟨9⟩ 2. QED
            BY (9) 1 DEF HashDomain
    ⟨8⟩ 4. newStateHash ∈ HashDomain
        ⟨9⟩ 1. newStateHash ∈ HashType
            BY (4) 2
        ⟨9⟩ 2. QED
            BY (9) 1 DEF HashDomain
    ⟨8⟩ 5. QED
        BY (6) 1, (8) 1, (8) 2, (8) 3, (8) 4, HashCollisionResistant
        DEF cardHistoryStateBinding, newHistoryStateBinding
    ⟨8⟩ 2. LL1NVRAM.historySummary′ = newHistorySummary
        ⟨8⟩ 1. LL1NVRAM′ = [historySummary ↦ newHistorySummary, symmetricKey ↦ LL1NVRAM.symmetricKey]
            BY (4) 1 DEF newHistorySummary, newPrivateStateEnc, sResult, privateState
        ⟨8⟩ 2. QED
            BY (8) 1
Since the history summaries are equal, their hash cardinalities are equal.

\[ \text{HashCardinality}(\text{historySummary}) = \text{HashCardinality}(\text{LL1NVRAM.historySummary'}) \]

We then have to prove that the hash cardinalities are natural numbers, to enable the prover to conclude that the equality above implies the less-than-or-equal we are trying to prove.

\[ \text{HashCardinality}(\text{historySummary}) \in \text{Nat} \]

\[ \text{LL1NVRAM.historySummary' } \in \text{HashType} \]

\[ \text{LL1NVRAM.historySummary' } \in \text{HashDomain} \]

The two cases are exhaustive, so the CardinalityInvariant is proven.

Third, we'll prove the UniquenessInvariant in the primed state.

To prove the universally quantified expression, we take a new set of variables of the appropriate types. For the take step to be meaningful to the prover, first we have to tell the prover to expand the definition of UniquenessInvariant, so it will see the universally quantified expression therein.

\[ \text{UniquenessInvariant'} \]

To simplify the writing of the proof, we re-state the definitions from the UniquenessInvariant.

\[ \text{uniqHistoryStateBinding1 } \triangleq \text{Hash(LL1NVRAM.historySummary, stateHash1)} \]

\[ \text{uniqHistoryStateBinding2 } \triangleq \text{Hash(LL1NVRAM.historySummary, stateHash2)} \]

The UniquenessInvariant states an implication. To prove this, it suffices to assume the antecedent and prove the consequent.

\[ \text{assume } \land \text{LL1HistoryStateBindingAuthenticated(uniqHistoryStateBinding1') } \land \text{LL1HistoryStateBindingAuthenticated(uniqHistoryStateBinding2')} \]

\[ \text{prove stateHash1 = stateHash2} \]

We hide the definitions of UniquenessInvariant and the definitions from the UniquenessInvariant.

The proof of UniquenessInvariant' employs CardinalityInvariant. To facilitate our extremely simple arithmetic, we begin by proving the types of a couple of critical hash cardinalities.

\[ \text{HashCardinality}(\text{LL1NVRAM.historySummary}) \in \text{Nat} \]

\[ \text{LL1NVRAM.historySummary' } \in \text{HashDomain} \]
We'll prove that the hash cardinality of the history summary in the $LL1NVRAM$ increases when a $LL1PerformOperation$ action occurs. This follows from the fact that the hash cardinality of the history summary in the NVRAM increases by one when a $LL1PerformOperation$ action occurs. This is straightforward though somewhat tedious.

First, we prove a bunch of types that are needed by the hash cardinality assumptions or for proving basic arithmetic.

With the type statements out of the way, we can construct a simple inequality. We do this in four linear steps.
Next, we’ll prove that the primed history state binding defined by the UniquenessInvariant is equal to the new state history state binding defined by the LL1PerformOperation action.

(7.1) \( LL1NVRAM . historySummary' = Hash( LL1NVRAM . historySummary, input1) \)

(8.1) newHistorySummary = Hash( LL1NVRAM . historySummary, input1) 

by Def newHistorySummary

(8.2) LL1NVRAM . historySummary' = newHistorySummary 

(9.1) LL1NVRAM' = [historySummary \mapsto newHistorySummary, 

symmetricKey \mapsto LL1NVRAM . symmetricKey] 

by (4) 1 Def newHistorySummary, newPrivateStateEnc, sResult, privateState 

(9.2) QED 

by (9) 1

(8.3) QED 

by (8) 1, (8) 2

(7.2) QED 

by (7) 1, (6) 4

(6) 6. HashCardinality( LL1NVRAM . historySummary') = HashCardinality( LL1NVRAM . historySummary) + 1 

(7.3) HashCardinality( input1) = 0 

(8.1) input1 \in InputType 

(9.1) input1 \in LL1AvailableInputs 

by (4) 1 

(9.2) LL1AvailableInputs \subseteq InputType 

(10.1) L1TypeInvariant 

by (2) 1 

(10.2) QED 

by (10) 1 Def L1TypeInvariant 

(9.3) QED 

by (9) 1, (9) 2 

(8.2) QED 

by (8) 1, InputCardinalityZero 

(7.4) QED 

by (5) 4, (5) 5, (6) 5, (7) 3 

(6) 7. QED 

by (5) 4, (5) 5, (6) 6

We start by proving that either the history state binding is authenticated in the unprimed state or the history state binding is authenticated by the new authenticator defined by the LL1PerformOperation action. This follows from the fact that the primed set of authenticators is constructed as the union of the unprimed set and the new authenticator.

(7.1) \( LL1HistoryStateBindingAuthenticated(uniqHistoryStateBinding1') \) 

\( \lor \) ValidateMAC( LL1NVRAM . symmetricKey, uniqHistoryStateBinding1', newAuthenticator) 

(8.1) LL1HistoryStateBindingAuthenticated(uniqHistoryStateBinding1') 

by (5) 3 

(8.2) UNCHANGED LL1NVRAM . symmetricKey 

by (2) 1, SymmetricKeyConstantLemma 

(8.3) LL1ObservedAuthenticators' = 

LL1ObservedAuthenticators \cup \{newAuthenticator\} 

by (4) 1 Def newAuthenticator, newHistoryStateBinding, newStateHash, 

newHistorySummary, newPrivateStateEnc, sResult, privateState 

(8.4) QED 

by (8) 1, (8) 2, (8) 3 Def LL1HistoryStateBindingAuthenticated
We then prove that the history state binding is not authenticated in the unprimed state. We use proof by contradiction. We show that if the history state binding were authenticated in the unprimed state, we could conclude a hash cardinality inequality that contradicts the hash cardinality inequality we proved above.

\[ (7.2) \quad \neg LL1HistoryStateBindingAuthenticated(uniqHistoryStateBinding1') \]

\[ \langle 8 \rangle 1. \text{SUFFICES} \]

\[ \quad \text{ASSUME } LL1HistoryStateBindingAuthenticated(uniqHistoryStateBinding1') \]

\[ \quad \text{PROVE } \text{FALSE} \]

\[ \quad \text{OBVIOUS} \]

\[ \langle 8 \rangle 2. \quad \text{HashCardinality}(LL1NVRAM.historySummary') \leq \]

\[ \quad \text{HashCardinality}(LL1NVRAM.historySummary) \]

\[ \langle 9 \rangle 1. \quad \text{CardinalityInvariant} \]

\[ \quad \text{BY } (2) 1 \]

\[ \langle 9 \rangle 2. \quad LL1NVRAM.historySummary' \in \text{HashType} \]

\[ \quad \text{BY } (4) 4 \]

\[ \langle 9 \rangle 3. \quad \text{stateHash1} \in \text{HashType} \]

\[ \quad \text{BY } (5) 2 \]

\[ \langle 9 \rangle 4. \quad LL1NVRAMHistorySummaryUncorrupted \]

\[ \quad \text{BY } (4) 5 \]

\[ \langle 9 \rangle 5. \text{QED} \]

\[ \quad \text{BY } (8) 1, (9) 1, (9) 2, (9) 3, (9) 4 \text{ DEF CardinalityInvariant, uniqHistoryStateBinding1} \]

\[ \langle 8 \rangle 3. \text{QED} \]

\[ \quad \text{BY } (5) 4, (5) 5, (5) 6, (8) 2, \text{ GEQorLT} \]

\[ \langle 7 \rangle 3. \text{QED} \]

\[ \quad \text{BY } (7) 1, (7) 2 \]

Since the new authenticator authenticates the new history state binding defined in the \textit{LL1PerformOperation} action, it follows that the history state bindings are equal, by the \textit{MACCollisionResistant} property.

We first have to prove some basic types, then we can appeal to the \textit{MACCollisionResistant} property.

\[ \langle 6 \rangle 2. \quad LL1NVRAM.symmetricKey \in \text{SymmetricKeyType} \]

\[ \quad \text{BY } (4) 3 \]

\[ \langle 6 \rangle 3. \quad uniqHistoryStateBinding1' \in \text{HashType} \]

\[ \langle 7 \rangle 1. \quad LL1NVRAM.historySummary' \in \text{HashDomain} \]

\[ \quad \langle 8 \rangle 1. \quad LL1NVRAM.historySummary' \in \text{HashType} \]

\[ \quad \text{BY } (2) 1 \]

\[ \langle 9 \rangle 2. \text{QED} \]

\[ \quad \text{BY } (9) 1, \text{ LL1SubtypeImplicationLemmaDef LL1SubtypeImplication} \]

\[ \langle 8 \rangle 2. \text{QED} \]

\[ \quad \text{BY } (8) 1 \text{ DEF HashDomain} \]

\[ \langle 7 \rangle 2. \quad \text{stateHash1} \in \text{HashDomain} \]

\[ \quad \langle 8 \rangle 1. \quad \text{stateHash1} \in \text{HashType} \]

\[ \quad \text{BY } (5) 2 \]

\[ \langle 8 \rangle 2. \text{QED} \]

\[ \quad \text{BY } (3) 1 \text{ DEF HashDomain} \]

\[ \langle 7 \rangle 3. \text{QED} \]

\[ \quad \text{BY } (7) 1, (7) 2, \text{ HashTypeSafeDef uniqHistoryStateBinding1} \]

\[ \langle 6 \rangle 4. \quad \text{newHistoryStateBinding} \in \text{HashType} \]

\[ \quad \text{BY } (4) 2 \]

\[ \langle 6 \rangle 5. \text{QED} \]

\[ \quad \text{BY } (6) 1, (6) 2, (6) 3, (6) 4, \text{ MACCollisionResistantDef newAuthenticator} \]

Next, we’ll prove that the primed history state binding 2 defined by the \textit{UniquenessInvariant} is equal to the new state history state binding defined by the \textit{LL1PerformOperation} action.

\[ (5) 8. \quad uniqHistoryStateBinding2' = \text{newHistoryStateBinding} \]
We first have to prove some basic types, then we can appeal to the MACCollisionResistant property. Since the new authenticator authenticates the new history state binding defined in the \textit{LL1PerformOperation} action. This follows from the fact that the primed set of authenticators is constructed as the union of the unprimed set and the new authenticator.

We then prove that the history state binding is not authenticated in the unprimed state. We use proof by contradiction. We show that if the history state binding were authenticated in the unprimed state, we could conclude a hash cardinality inequality that contradicts the hash cardinality inequality we proved above.

We then prove that the history state binding is not authenticated in the unprimed state. We use proof by contradiction. We show that if the history state binding were authenticated in the unprimed state, we could conclude a hash cardinality inequality that contradicts the hash cardinality inequality we proved above.
BY (9)1, **LL1SubtypeImplicationLemma**\(\text{def} \ LL1SubtypeImplication\)
\(\langle 8\rangle 2. \text{QED} \)
BY (8)1 **def** HashDomain
\(\langle 7\rangle 2. \text{stateHash}2 \in \text{HashDomain} \)
\(\langle 8\rangle 1. \text{stateHash}2 \in \text{HashType} \)
BY (5)2
\(\langle 8\rangle 2. \text{QED} \)
BY (3)1 **def** HashDomain
\(\langle 7\rangle 3. \text{QED} \)
BY (7)1, (7)2, **HashTypeSafe**\(\text{def} \ \text{uniqHistoryStateBinding}2 \)
\(\langle 6\rangle 4. \text{newHistoryStateBinding} \in \text{HashType} \)
BY (4)2
\(\langle 6\rangle 5. \text{QED} \)
BY (6)1, (6)2, (6)3, (6)4, **MACCollisionResistant**\(\text{def} \ \text{newAuthenticator} \)

Since each of the history state bindings is equal to the history state binding defined by the **LL1PerformOperation** action, the two history state bindings must be equal to each other.

\(\langle 5\rangle 9. \ \text{uniqHistoryStateBinding}1' = \text{uniqHistoryStateBinding}2' \)
BY (5)7, (5)8

Since the inputs-state bindings are equal, it follows that the state hashes are equal, by the **HashCollisionResistant** property.

\(\langle 5\rangle 10. \text{QED} \)
\(\langle 6\rangle 1. \ \text{LL1NVRAM}.\text{historySummary}' \in \text{HashDomain} \)
\(\langle 7\rangle 1. \ \text{LL1NVRAM}.\text{historySummary}' \in \text{HashType} \)
BY (4)4
\(\langle 7\rangle 2. \text{QED} \)
BY (7)1 **def** HashDomain
\(\langle 6\rangle 2. \text{stateHash}1 \in \text{HashDomain} \)
\(\langle 7\rangle 1. \text{stateHash}1 \in \text{HashType} \)
BY (5)2
\(\langle 7\rangle 2. \text{QED} \)
BY (7)1 **def** HashDomain
\(\langle 6\rangle 3. \text{stateHash}2 \in \text{HashDomain} \)
\(\langle 7\rangle 1. \text{stateHash}2 \in \text{HashType} \)
BY (5)2
\(\langle 7\rangle 2. \text{QED} \)
BY (7)1 **def** HashDomain
\(\langle 6\rangle 4. \text{QED} \)
BY (5)9, (6)1, (6)2, (6)3, **HashCollisionResistant**
**def** uniqHistoryStateBinding1, uniqHistoryStateBinding2
\(\langle 4\rangle 9. \text{QED} \)
BY (4)6, (4)7, (4)8

The **LL1RepeatOperation** case is a straightforward application of the **InclusionUnchangedLemma**, the **CardinalityUnchangedLemma**, and the **UniquenessUnchangedLemma**, the preconditions for which follow from the **LL1RepeatOperationUnchangedObservedOutputsLemma** and the **LL1RepeatOperationUnchangedAuthenticatedHistoryStateBindingsLemma**.

\(\langle 3\rangle 3. \text{CASE} \ \text{LL1RepeatOperation} \)
\(\langle 4\rangle 1. \text{PIECE} \ \text{input} \in \text{LL1AvailableInputs} : \text{LL1RepeatOperation}!\text{(input)}!1 \)
BY (3)3 **def** **LL1RepeatOperation**
\(\langle 4\rangle 2. \text{UNCHANGED} \ \text{LL1NVRAM} \)
BY (4)1
\(\langle 4\rangle 3. \text{UNCHANGED} \ \text{LL1ObservedOutputs} \)
\(\langle 5\rangle 1. \text{LL1TypeInvariant} \land \text{UnforgeabilityInvariant} \land \text{InclusionInvariant} \)
1. \( \text{CardinalityUnchangedLemma} \)

2. \( \text{UniquenessInvariant} \)

3. \( \text{InclusionInvariant} \)

4. \( \forall \text{historyStateBinding} \in \text{HashType} : \)

   \( \text{UNCHANGED} \ \text{LL1HistoryStateBindingAuthenticated} \ (\text{historyStateBinding}) \)

5. \( \text{LL1TypeInvariant} \land \text{UnforgeabilityInvariant} \land \text{InclusionInvariant} \)

   \( \text{cards} \)

6. \( \text{qed} \)

   \( \text{cards} \)

   \( \text{cards} \)

7. \( \text{cards} \)

8. \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)

   \( \text{cards} \)
(4) 6. QED
   by (4) 3, (4) 4, (4) 5

The $LL1ReadDisk$ case is a straightforward application of the $InclusionUnchangedLemma$, the $CardinalityUnchangedLemma$, and the $UniquenessUnchangedLemma$.

(3) 5. CASE $LL1ReadDisk$
(4) 1. UNCHANGED \{ $LL1NVRAM$, $LL1ObservedOutputs$ \}
   by (3) 5 DEF $LL1ReadDisk$
(4) 2. $\forall$ historyStateBinding $\in$ HashType :
       UNCHANGED $LL1HistoryStateBindingAuthenticated(historyStateBinding)$
       (5) 1. UNCHANGED \{ $LL1NVRAM$, $LL1ObservedAuthenticators$ \}
           by (3) 5 DEF $LL1ReadDisk$
       (5) 2. QED
           by (5) 1, $UnchangedAuthenticatedHistoryStateBindingsLemma$
(4) 3. $InclusionInvariant'$
   (5) 1. $InclusionInvariant$ $\land$ $LL1TypeInvariant$ $\land$ $LL1TypeInvariant'$
       by (2) 1
   (5) 2. QED
       by (4) 1, (4) 2, (5) 1, $InclusionUnchangedLemma$
(4) 4. $CardinalityInvariant'$
   (5) 1. $CardinalityInvariant$ $\land$ $LL1TypeInvariant$
       by (2) 1
   (5) 2. QED
       by (4) 1, (4) 2, (5) 1, $CardinalityUnchangedLemma$
(4) 5. $UniquenessInvariant'$
   (5) 1. $UniquenessInvariant$ $\land$ $LL1TypeInvariant$
       by (2) 1
   (5) 2. QED
       by (4) 1, (4) 2, (5) 1, $UniquenessUnchangedLemma$
(4) 6. QED
       by (4) 3, (4) 4, (4) 5

The $LL1WriteDisk$ case is a straightforward application of the $InclusionUnchangedLemma$, the $CardinalityUnchangedLemma$, and the $UniquenessUnchangedLemma$.

(3) 6. CASE $LL1WriteDisk$
(4) 1. UNCHANGED \{ $LL1NVRAM$, $LL1ObservedOutputs$ \}
   by (3) 6 DEF $LL1WriteDisk$
(4) 2. $\forall$ historyStateBinding $\in$ HashType :
       UNCHANGED $LL1HistoryStateBindingAuthenticated(historyStateBinding)$
       (5) 1. UNCHANGED \{ $LL1NVRAM$, $LL1ObservedAuthenticators$ \}
           by (3) 6 DEF $LL1WriteDisk$
       (5) 2. QED
           by (5) 1, $UnchangedAuthenticatedHistoryStateBindingsLemma$
(4) 3. $InclusionInvariant'$
   (5) 1. $InclusionInvariant$ $\land$ $LL1TypeInvariant$ $\land$ $LL1TypeInvariant'$
       by (2) 1
   (5) 2. QED
       by (4) 1, (4) 2, (5) 1, $InclusionUnchangedLemma$
(4) 4. $CardinalityInvariant'$
   (5) 1. $CardinalityInvariant$ $\land$ $LL1TypeInvariant$
       by (2) 1
   (5) 2. QED
       by (4) 1, (4) 2, (5) 1, $CardinalityUnchangedLemma$
(4) 5. $UniquenessInvariant'$
The LL1CorruptRAM case is a straightforward application of the InclusionUnchangedLemma, the CardinalityUnchangedLemma, and the UniquenessUnchangedLemma.

The LL1RestrictedCorruption case is non-trivial, although nowhere near as involved as the LL1PerformOperation case.

We pick a garbage history summary in the appropriate type.

The primed value of the history summary in the NVRAM equals this garbage history summary.

We now prove each invariant separately, starting with the InclusionInvariant.
To prove the universally quantified expression, we take a new set of variables of the appropriate types. For the \textsc{take} step to be meaningful to the prover, first we have to tell the prover to expand the definition of \textit{InclusionInvariant}, so it will see the universally quantified expression therein.

(5) \textsc{use def} \textit{InclusionInvariant}

(5)1. \textsc{take} \textit{input} \in \textit{InputType},
\textit{historySummary} \in \textit{HashType},
\textit{publicState} \in \textit{PublicStateType},
\textit{privateStateEnc} \in \textit{PrivateStateEncType}

To simplify the writing of the proof, we re-state the definitions from the \textit{InclusionInvariant}.

(5) incl\textit{StateHash} \triangleright= \textit{Hash}(\textit{publicState}, \textit{privateStateEnc})
(5) incl\textit{HistoryStateBinding} \triangleright= \textit{Hash}(\textit{historySummary}, incl\textit{StateHash})
(5) incl\textit{PrivateState} \triangleright= \textit{SymmetricDecrypt}(\textit{LL1NVRAM}.\textit{symmetricKey}, \textit{privateStateEnc})
(5) incl\textit{SResult} \triangleright= \textit{Service}\textit{(publicState, inclPrivateState, input)}
(5) incl\textit{NewPrivateStateEnc} \triangleright= \textit{SymmetricEncrypt}(\textit{LL1NVRAM}.\textit{symmetricKey}, incl\textit{SResult}.new\textit{PrivateState})
(5) incl\textit{NewStateHash} \triangleright= \textit{Hash}(incl\textit{SResult}.new\textit{PublicState}, incl\textit{NewPrivateStateEnc})
(5) incl\textit{NewHistoryStateBinding} \triangleright= \textit{Hash}(\textit{LL1NVRAM}.\textit{historySummary}, incl\textit{NewStateHash})

We then assert the type safety of these definitions, with the help of the \textit{InclusionInvariantDefsTypeSafeLemma}.

(5)2. \& incl\textit{StateHash} \in \textit{HashType}
\& incl\textit{HistoryStateBinding} \in \textit{HashType}
\& incl\textit{PrivateState} \in \textit{PrivateStateType}
\& incl\textit{SResult} \in \textit{ServiceResultType}
\& incl\textit{SResult}.new\textit{PublicState} \in \textit{PublicStateType}
\& incl\textit{SResult}.new\textit{PrivateState} \in \textit{PrivateStateType}
\& incl\textit{SResult}.output \in \textit{OutputType}
\& incl\textit{NewPrivateStateEnc} \in \textit{PrivateStateEncType}
\& incl\textit{NewStateHash} \in \textit{HashType}
\& incl\textit{NewHistoryStateBinding} \in \textit{HashType}

(6)1. \textit{LL1TypeInvariant}
\textsc{by} (2)1

(6)2. \textsc{QED}
\textsc{by} (5)1, (6)1, \textit{InclusionInvariantDefsTypeSafeLemma}

The \textit{InclusionInvariant} states an implication. To prove this, it suffices to prove that the the antecedent is false. The antecedent is a conjunction. We will prove that the first conjunct implies that the second conjunct is false.

(5)3. \textsc{suffices}
\textsc{assume true}
\textsc{prove}
\textit{LL1NVRAM}.\textit{historySummary}' = \textit{Hash}(\textit{historySummary}, \textit{input}) \Rightarrow
\neg\textit{LL1HistoryStateBindingAuthenticated}(incl\textit{HistoryStateBinding})'

\textsc{obvious}

We hide the definition of \textit{InclusionInvariant} and the definitions from the \textit{InclusionInvariant}.

(5) \textsc{hide def} \textit{InclusionInvariant}

(5) \textsc{hide def} incl\textit{StateHash}, incl\textit{HistoryStateBinding}, incl\textit{PrivateState}, incl\textit{SResult},
incl\textit{NewPrivateStateEnc}, incl\textit{NewStateHash}, incl\textit{NewHistoryStateBinding}

We assume the antecedent in this conjunction.

(5)4. \textsc{have} \textit{LL1NVRAM}.\textit{historySummary}' = \textit{Hash}(\textit{historySummary}, \textit{input})

We prove that all of the authenticators in the set of observed authenticators fail to validate the history state binding defined by the inclusion invariant in the unprimed state.

(5)5. \forall \textit{authenticator} \in \textit{LL1ObservedAuthenticators}:
\neg\textsc{ValidateMAC}(\textit{LL1NVRAM}.\textit{symmetricKey}, incl\textit{HistoryStateBinding}, \textit{authenticator})
We will make use of the conjunct in $LL_1\text{RestrictedCorruption}$ that prevents the garbage history summary from being a predecessor to any history summary in an authenticated history state binding. This conjunct states a 4-way universally quantified predicate.

$$\langle 6 \rangle 1. LL_1\text{RestrictedCorruption} \neg \text{nvramprevious} (\text{garbageHistorySummary})$$

BY $\langle 4 \rangle 1$

We will pare the predicate down to a single universal quantification by showing particular instances for three of the quantifiers.

$$\langle 6 \rangle 2. \text{inclStateHash} \in \text{HashType}$$

BY $\langle 5 \rangle 2$

$$\langle 6 \rangle 3. \text{historySummary} \in \text{HashType}$$

BY $\langle 5 \rangle 1$

$$\langle 6 \rangle 4. \text{input} \in \text{InputType}$$

BY $\langle 5 \rangle 1$

We also need to show that the antecedent of the implication in this predicate is satisfied.

$$\langle 6 \rangle 5. \text{garbageHistorySummary} = \text{Hash} (\text{historySummary}, \text{input})$$

BY $\langle 4 \rangle 2, \langle 5 \rangle 4$

The consequent of the implication in this predicate is exactly what we need for our conclusion.

$$\langle 6 \rangle 6. \text{qed}$$

BY $\langle 6 \rangle 1, \langle 6 \rangle 2, \langle 6 \rangle 3, \langle 6 \rangle 4, \langle 6 \rangle 5$ DEF $\text{inclHistoryStateBinding}$

We show that the symmetric key in the NVRAM has not changed, nor has the set of observed authenticators.

$$\langle 5 \rangle 6. \text{UNCHANGED} LL_1\text{NVRAM}.\text{symmetricKey}$$

BY $\langle 2 \rangle 1, \text{SymmetricKeyConstantLemma}$

$$\langle 5 \rangle 7. \text{UNCHANGED} LL_1\text{ObservedAuthenticators}$$

BY $\langle 3 \rangle 8$ DEF $LL_1\text{RestrictedCorruption}$

Thus, we can conclude that all of the authenticators in the set of observed authenticators fail to validate the history state binding defined by the inclusion invariant in the primed state. This satisfies the definition of $LL_1\text{HistoryStateBindingAuthenticated}$.

$$\langle 5 \rangle 8. \text{QED}$$

BY $\langle 5 \rangle 5, \langle 5 \rangle 6, \langle 5 \rangle 7$ DEF $LL_1\text{HistoryStateBindingAuthenticated}$

We next prove the $CardinalityInvariant'$.

$$\langle 4 \rangle 4. \text{CardinalityInvariant}'$$

To prove the universally quantified expression, we take a new set of variables of the appropriate types. For the take step to be meaningful to the prover, first we have to tell the prover to expand the definition of $CardinalityInvariant'$, so it will see the universally quantified expression therein.

$$\langle 5 \rangle \text{USE DEF} \text{CardinalityInvariant}$$

$$\langle 5 \rangle 1. \text{TAKEx historySummary} \in \text{HashType}, \text{stateHash} \in \text{HashType}$$

The $CardinalityInvariant$ states an implication. To prove this, it suffices to prove that the antecedent is false. The antecedent is a conjunction. We will prove that the first conjunct is false.

$$\langle 5 \rangle 2. \text{SUFFICES ASSUME TRUE PROVE} \neg LL_1\text{NVRAMHistorySummaryUncorrupted}'$$

OBVIOUS

We then hide the definition.

$$\langle 5 \rangle \text{HIDE DEF} \text{CardinalityInvariant}$$

We will make use of the conjunct in $LL_1\text{RestrictedCorruption}$ that prevents the garbage history summary from being in an authenticated history state binding.

$$\langle 5 \rangle 3. LL_1\text{RestrictedCorruption} \neg \text{nvrampcurrent} (\text{garbageHistorySummary})$$

BY $\langle 4 \rangle 1$

The following equivalence, plus the knowledge that the symmetric key in the NVRAM and the set of observed authenticators have not changed, are sufficient to prove the conclusion.

$$\langle 5 \rangle 4. LL_1\text{NVRAM.\text{historySummary}}' = \text{garbageHistorySummary}$$

BY $\langle 4 \rangle 2$
(5) 5. UNCHANGED $LL1NVRAM$.symmetricKey
    BY ⟨2⟩1, SymmetricKeyConstantLemma
(5) 6. UNCHANGED $LL1$ObservedAuthenticators
    BY ⟨3⟩8 DEF $LL1$RestrictedCorruption
(5) 7. QED
    BY ⟨5⟩3, ⟨5⟩4, ⟨5⟩5, ⟨5⟩6
    DEF $LL1NVRAM$HistorySummaryUncorrupted, $LL1$HistoryStateBindingAuthenticated

We last prove the UniquenessInvariant.

(4) 5. UniquenessInvariant
    To prove the universally quantified expression, we take a new set of variables of the appropriate types. For
    the take step to be meaningful to the prover, first we have to tell the prover to expand the definition of
    UniquenessInvariant, so it will see the universally quantified expression therein.

(5) USE DEF UniquenessInvariant
(5) 1. TAKE stateHash1, stateHash2 ∈ HashType
    To simplify the writing of the proof, we re-state the definitions from the UniquenessInvariant.

(5) uniqHistoryStateBinding1 ≜ $Hash($LL1NVRAM$.historySummary, stateHash1$)
(5) uniqHistoryStateBinding2 ≜ $Hash($LL1NVRAM$.historySummary, stateHash2$)

The UniquenessInvariant states an implication. To prove this, it suffices to prove that the antecedent is
false. The antecedent is a conjunction. We will prove that the first conjunct is false.

(5) 2. SUFFICES
    ASSUME TRUE
    PROVE ¬$LL1$HistoryStateBindingAuthenticated(uniqHistoryStateBinding1)'
    OBVIOUS

We hide the definitions of UniquenessInvariant and the definitions from the UniquenessInvariant.

(5) HIDE DEF UniquenessInvariant
(5) HIDE DEF uniqHistoryStateBinding1, uniqHistoryStateBinding2

We will make use of the conjunct in $LL1$RestrictedCorruption that prevents the garbage history summary from
being in an authenticated history state binding. This conjunct states a 2-way universally quantified predicate.

(5) 3. $LL1$RestrictedCorruption!nvram!current(garbageHistorySummary)
    BY ⟨4⟩1

We will pare the predicate down to a single universal quantification by showing particular instances for one of
the quantifiers.

(5) 4. stateHash1 ∈ HashType
    BY ⟨5⟩1

The following equivalence, plus the knowledge that the symmetric key in the NVRAM and the set of observed
authenticators have not changed, are sufficient to prove the conclusion.

(5) 5. $LL1NVRAM$.historySummary' = garbageHistorySummary
    BY ⟨4⟩2
(5) 6. UNCHANGED $LL1NVRAM$.symmetricKey
    BY ⟨2⟩1, SymmetricKeyConstantLemma
(5) 7. UNCHANGED $LL1$ObservedAuthenticators
    BY ⟨3⟩8 DEF $LL1$RestrictedCorruption
(5) 8. QED
    BY ⟨5⟩3, ⟨5⟩4, ⟨5⟩5, ⟨5⟩6, ⟨5⟩7
    DEF uniqHistoryStateBinding1, $LL1$HistoryStateBindingAuthenticated
(4) 6. QED
    BY ⟨4⟩3, ⟨4⟩4, ⟨4⟩5
(3) 9. QED
    BY ⟨2⟩3, ⟨3⟩1, ⟨3⟩2, ⟨3⟩3, ⟨3⟩4, ⟨3⟩5, ⟨3⟩6, ⟨3⟩7, ⟨3⟩8 DEF $LL1$Next
(2) 4. QED
    BY ⟨2⟩1, ⟨2⟩2, ⟨2⟩3
Using the Inv 1 proof rule, the base case and the induction step together imply that the invariant always holds.

\[\{1\}5. \text{QED}\]

From the above proof and the previous proof that the UnforgeabilityInvariant is an inductive invariant of the Memoir-Basic spec, it proves that the set of CorrectnessInvariants are inductive invariants of the Memoir-Basic spec.

THEOREM CorrectnessInvariance $\triangleq$ \textit{LL1Spec} $\Rightarrow$ $\square$CorrectnessInvariants

\[\{1\}1. \text{LL1Spec} \Rightarrow \square\text{UnforgeabilityInvariant}\]

\[\text{BY UnforgeabilityInvariance}\]

\[\{1\}2. \text{LL1Spec} \Rightarrow \square\text{InclusionInvariant} \land \square\text{UniquenessInvariant}\]

\[\text{BY InclusionCardinalityUniquenessInvariance}\]

\[\{1\}3. \text{UnforgeabilityInvariant} \land \text{InclusionInvariant} \land \text{UniquenessInvariant} \Rightarrow \text{CorrectnessInvariants}\]

\[\text{BY DEF CorrectnessInvariants}\]

\[\{1\}4. \text{QED}\]

\[\text{BY } \{1\}1, \{1\}2, \{1\}3\]
This module proves that the Memoir-Basic spec implements the high-level spec, under the defined refinement. It begins with two supporting lemmas:

NonAdvancementLemma

LL1NVRAMHistorySummaryUncorruptedEqualsHLAliveLemma

Then, it states and proves the LL1Implementation theorem.

The NonAdvancementLemma proves that, if there is no change to the NVRAM or to the authentication status of any history state binding, then the high-level public and private state defined by the refinement both stutter.

THEOREM NonAdvancementLemma ⊆

(∧ LL1Refinement
 ∧ LL1Refinement'
 ∧ LL1TypeInvariant
 ∧ LL1TypeInvariant'
 ∧ UniquenessInvariant
 ∧ UNCHANGED LL1NVRAM
 ∧ ∀ historyStateBinding ∈ HashType :
    UNCHANGED LL1HistoryStateBindingAuthenticated(historyStateBinding))

⇒ UNCHANGED ⟨HLPublicState, HLPrivateState⟩

We assume the antecedent.

⟨1⟩1. HAVE ∧ LL1Refinement
 ∧ LL1Refinement'
 ∧ LL1TypeInvariant
 ∧ LL1TypeInvariant'
 ∧ UniquenessInvariant
 ∧ UNCHANGED LL1NVRAM
 ∧ ∀ historyStateBinding ∈ HashType :
    UNCHANGED LL1HistoryStateBindingAuthenticated(historyStateBinding)

⟨1⟩2. LL1TypeInvariant ∧ LL1TypeInvariant'

by ⟨1⟩1

These definitions are copied from the LL1Refinement.

⟨1⟩ refPrivateStateEnc ≅ SymmetricEncrypt(LL1NVRAM, symmetricKey, HLPrivateState)
⟨1⟩ refStateHash ≅ Hash(HLPublicState, refPrivateStateEnc)
⟨1⟩ refHistoryStateBinding ≅ Hash(LL1NVRAM, historySummary, refStateHash)

We prove that the definitions satisfy their types in both the unprimed and primed states, using the LL1RefinementDefsTypeSafeLemma and the LL1RefinementPrimeDefsTypeSafeLemma.

⟨1⟩3. ∧ refPrivateStateEnc ∈ PrivateStateEncType
 ∧ refStateHash ∈ HashType
 ∧ refHistoryStateBinding ∈ HashType

by ⟨1⟩1, ⟨1⟩2, LL1RefinementDefsTypeSafeLemma

⟨1⟩4. ∧ refPrivateStateEnc' ∈ PrivateStateEncType
 ∧ refStateHash' ∈ HashType
 ∧ refHistoryStateBinding' ∈ HashType

by ⟨1⟩1, ⟨1⟩2, LL1RefinementPrimeDefsTypeSafeLemma

We then hide the definitions. We’ll pull them in as needed later.

⟨1⟩ HIDE DEF refPrivateStateEnc, refStateHash, refHistoryStateBinding
We consider the two possible states of the \text{LL1NVRAMHistorySummaryUncorrupted} predicate separately. In the first case, that the history state binding in the NVRAM is authenticated in the unprimed state.

\(\langle 1 \rangle 5.\) \text{CASE LL1NVRAMHistorySummaryUncorrupted}

We first prove that the history state binding in the NVRAM is authenticated in the primed state, as well. This will be needed a few places below.

\(\langle 2 \rangle 1.\) \text{LL1NVRAMHistorySummaryUncorrupted}'

\(\langle 3 \rangle 1.\) \text{LL1NVRAMHistorySummaryUncorrupted}

\by \langle 1 \rangle 5

\(\langle 3 \rangle 2.\) \text{UNCHANGED LL1NVRAMHistorySummaryUncorrupted}

\(\langle 4 \rangle 1.\) \text{LL1TypeInvariant}

\by \langle 1 \rangle 1

\(\langle 4 \rangle 2.\) QED

\by \langle 4 \rangle 1, \text{LL1NVRAMHistorySummaryUncorruptedUnchangedLemma}

\(\langle 3 \rangle 3.\) QED

\by \langle 3 \rangle 1, \langle 3 \rangle 2

We show that the refined public state does not change and that the refined encrypted private state does not change. We will use the \text{HashCollisionResistant} property.

\(\langle 2 \rangle 2.\) \text{UNCHANGED HLPublicState}

\by \langle 2 \rangle 1

\(\langle 3 \rangle 1.\) \text{HLPublicState} \in \text{HashDomain}

\(\langle 4 \rangle 1.\) \text{HLPublicState} \in \text{PublicStateType}

\by \langle 1 \rangle 1

\(\langle 5 \rangle 1.\) \text{LL1Refinement}

\by \langle 1 \rangle 1

\(\langle 5 \rangle 2.\) QED

\by \langle 1 \rangle 5, \langle 5 \rangle 1 \text{ DEF LL1Refinement}

\(\langle 4 \rangle 2.\) QED

\by \langle 4 \rangle 1 \text{ DEF HashDomain}

\(\langle 3 \rangle 2.\) \text{HLPublicState}' \in \text{HashDomain}

\by \langle 1 \rangle 1

\(\langle 4 \rangle 1.\) \text{HLPublicState}' \in \text{PublicStateType}

\by \langle 1 \rangle 1

\(\langle 5 \rangle 1.\) \text{LL1Refinement'}

\by \langle 1 \rangle 1

\(\langle 5 \rangle 2.\) QED

\by \langle 2 \rangle 1, \langle 5 \rangle 1 \text{ DEF LL1Refinement}

\(\langle 4 \rangle 2.\) QED

\by \langle 4 \rangle 1 \text{ DEF HashDomain}

\(\langle 3 \rangle 3.\) \text{refPrivateStateEnc} \in \text{HashDomain}

\by \langle 1 \rangle 3

\(\langle 4 \rangle 2.\) QED

\by \langle 4 \rangle 1 \text{ DEF HashDomain}

\(\langle 3 \rangle 4.\) \text{refPrivateStateEnc}' \in \text{HashDomain}

\by \langle 1 \rangle 4

\(\langle 4 \rangle 2.\) QED

\by \langle 4 \rangle 1 \text{ DEF HashDomain}

We then prove that the refined state hash does not change. We will use the \text{UniquenessInvariant}.

\(\langle 3 \rangle 5.\) \text{UNCHANGED refStateHash}
We prove some types we will need for the UniquenessInvariant.

1. \( \text{refStateHash} \in \text{HashType} \)
   BY (1):3

2. \( \text{refStateHash}' \in \text{HashType} \)
   BY (1):4

We prove that the unprimed history state binding is authenticated in the unprimed state. This follows directly from the LL1Refinement.

3. \( \text{LL1HistoryStateBindingAuthenticated}(\text{refHistoryStateBinding}) \)

Step 1: The primed history state binding is authenticated in the primed state. This follows directly from the primed LL1Refinement.

4. \( \exists \text{authenticator} \in \text{LL1ObservedAuthenticators}' : \text{ValidateMAC} (\text{LL1NVRAM.symmetricKey}', \text{refHistoryStateBinding}', \text{authenticator}) \)

Step 2: The primed history state binding is of HashType. We need this so we can apply the assumption that there is no change to the authentication status of any history state binding.

5. \( \text{refHistoryStateBinding}' \in \text{HashType} \)
   BY (1):4

Step 3: There is no change to the authentication status of any history state binding.

6. \( \forall \text{historyStateBinding} \in \text{HashType} : \)
   \( (\exists \text{authenticator} \in \text{LL1ObservedAuthenticators}' : \text{ValidateMAC} (\text{LL1NVRAM.symmetricKey}', \text{historyStateBinding}', \text{authenticator})) \)
   \( \Rightarrow (\exists \text{authenticator} \in \text{LL1ObservedAuthenticators} : \text{ValidateMAC} (\text{LL1NVRAM.symmetricKey}, \text{historyStateBinding}, \text{authenticator})) \)

Step 4: The primed history state binding is authenticated in the unprimed state.

7. \( \exists \text{authenticator} \in \text{LL1ObservedAuthenticators} : \text{ValidateMAC} (\text{LL1NVRAM.symmetricKey}, \text{refHistoryStateBinding}', \text{authenticator}) \)
   BY (5):1, (5):2, (5):3
(5)5. QED
  BY (5)4 DEF LL1HistoryStateBindingAuthenticated
Since the authentication status of the history state binding has not changed, the state hash has not changed.

(4)5. QED
  (5)1. UniquenessInvariant
  BY (1)1
(5)2. UNCHANGED LL1NVRAM.historySummary
  (6)1. UNCHANGED LL1NVRAM
  BY (1)1
  (6)2. QED
  BY (6)1
(5)3. QED
  BY (4)1, (4)2, (4)3, (4)4, (5)1, (5)2
  DEF UniquenessInvariant, refHistoryStateBinding
Since the refined state hash does not change, the refined public state and encrypted private state do not change, because the hash is collision-resistant.

(3)6. QED
  BY (3)1, (3)2, (3)3, (3)4, (3)5, HashCollisionResistantDEF refStateHash
From the previous step, it immediately follows that the refined public state does not change.

(2)3. UNCHANGED HLPublicState
  BY (2)2
It is slightly more involved to show that the refined private state does not change. We need to show that the symmetric key in the NVRAM does not change, and we also have to employ the correctness of the symmetric crypto operations.

(2)4. UNCHANGED HLPrivateState
  In the unprimed state, the refined private state equals the decryption of the refined encrypted private state, using the unprimed symmetry key in the NVRAM. This follows because the refined encrypted private state is defined in the LL1Refinement as the encryption of the refined private state, and the symmetric crypto operations are assumed to be correct.

(3)1. HLPrivateState = SymmetricDecrypt(LL1NVRAM.symmetricKey, refPrivateStateEnc)
  (4)1. LL1NVRAM.symmetricKey ∈ SymmetricKeyType
  BY (1)2, LL1SubtypeImplicationLemmaDEF LL1SubtypeImplication
  (4)2. HLPrivateState ∈ PrivateStateType
  (5)1. LL1Refinement
  BY (1)1
  (5)2. QED
  BY (1)5, (5)1 DEF LL1Refinement
  (4)3. QED
  BY (4)1, (4)2, SymmetricCryptoCorrectDEF refPrivateStateEnc
We then show that in the primed state, the refined private state equals the decryption of the refined encrypted private state, using the unprimed symmetry key in the NVRAM.

(3)2. HLPrivateState' = SymmetricDecrypt(LL1NVRAM.symmetricKey, refPrivateStateEnc)
  In the primed state, the refined private state equals the decryption of the refined encrypted private state, using the primed symmetry key in the NVRAM. This follows because the refined encrypted private state is defined in the LL1Refinement as the encryption of the refined private state, and the symmetric crypto operations are assumed to be correct.

(4)1. HLPrivateState' = SymmetricDecrypt(LL1NVRAM.symmetricKey', refPrivateStateEnc')
  (5)1. LL1NVRAM.symmetricKey' ∈ SymmetricKeyType
  BY (1)2, LL1SubtypeImplicationLemmaDEF LL1SubtypeImplication
  (5)2. HLPrivateState' ∈ PrivateStateType
  (6)1. LL1Refinement'
  BY (1)1
The symmetry key in the NVRAM is unchanged, and the refined encrypted private state is unchanged.

Since the relevant variables are unchanged, we can conclude that, in the primed state, the refined private state equals the decryption of the refined encrypted private state, using the unprimed symmetry key in the NVRAM.

Since both the unprimed and primed refined private state are equal to the same expression, they are equal to each other.

In the second case, the history state binding in the NVRAM is not authenticated in the unprimed state.

By the LL1Refinement, the high-level public and private states are equal to their dead states.

The history state binding in the NVRAM is also not authenticated in the primed state, so again by the LL1Refinement, the high-level public and private states are equal to their dead states.

Since both the unprimed and primed refined private state are equal to the same expression, they are equal to each other.

In the second case, the history state binding in the NVRAM is not authenticated in the unprimed state.

By the LL1Refinement, the high-level public and private states are equal to their dead states.

The history state binding in the NVRAM is also not authenticated in the primed state, so again by the LL1Refinement, the high-level public and private states are equal to their dead states.

Since both the unprimed and primed refined private state are equal to the same expression, they are equal to each other.
This simple lemma proves that the two predicates $LL1\text{NVRAMHistorySummaryUncorrupted}$ and $HL\text{Alive}$ are equal whenever $LL1\text{Refinement}$ is true, in either unprimed or primed state.

**THEOREM** $LL1\text{NVRAMHistorySummaryUncorruptedEqualsHLAliveLemma} \triangleq$

\[\begin{align*}
\land & \quad LL1\text{Refinement} \Rightarrow LL1\text{NVRAMHistorySummaryUncorrupted} = HL\text{Alive} \\
\land & \quad LL1\text{Refinement'} \Rightarrow LL1\text{NVRAMHistorySummaryUncorrupted'} = HL\text{Alive'}
\end{align*}\]

\(\langle 1\rangle 1.\) $LL1\text{Refinement} \Rightarrow LL1\text{NVRAMHistorySummaryUncorrupted} = HL\text{Alive}$

\(\langle 2\rangle 1.\) **HAVE** $LL1\text{Refinement}$

\(\langle 2\rangle 2.\) **IF** $LL1\text{NVRAMHistorySummaryUncorrupted} \text{ then } HL\text{Alive} = \text{TRUE ELSE } HL\text{Alive} = \text{FALSE}$

\(\langle 2\rangle 3.\) **IF** $LL1\text{NVRAMHistorySummaryUncorrupted'} \text{ then } HL\text{Alive'} = \text{TRUE ELSE } HL\text{Alive'} = \text{FALSE}$

\(\langle 2\rangle 4.\) **QED**

\(\langle 2\rangle 5.\) **QED**

\(\langle 1\rangle 2.\) **QED**

\(\langle 1\rangle 3.\) **QED**

\(\langle 1\rangle 7.\) **QED**

\(\langle 1\rangle 5, \langle 1\rangle 6\)

The $LL1\text{Implementation}$ theorem is where the rubber meets the road. This is the ultimate proof that the Memoir-Basic spec implements the high-level spec, under the defined refinement.

**THEOREM** $LL1\text{Implementation} \triangleq LL1\text{Spec} \land \Box LL1\text{Refinement} \Rightarrow HL\text{Spec}$

This proof will require the $LL1\text{TypeInvariant}$ and the $CorrectnessInvariants$. Fortunately, the $LL1\text{TypeSafe}$ theorem has already proven that the Memoir-Basic spec satisfies its type invariant, and the $CorrectnessInvariance$ theorem has already proven that the Memoir-Basic spec satisfies the $CorrectnessInvariant$.

\(\langle 1\rangle 1.\) $LL1\text{Spec} \Rightarrow \Box LL1\text{TypeInvariant}$

\(\langle 1\rangle 2.\) $LL1\text{Spec} \Rightarrow \Box CorrectnessInvariants$

\(\langle 1\rangle 3.\) $LL1\text{Init} \land LL1\text{Refinement} \land LL1\text{TypeInvariant} \land CorrectnessInvariants \Rightarrow HL\text{Init}$
We begin the base case by assuming the antecedent.

\( 2.1. \text{H} \\text{AVE} ~ L1Init \land L1Refinement \land L1TypeInvariant \land \text{CorrectnessInvariants} \)
\( 2.2. ~ L1TypeInvariant \)

\( \text{BY} \ (2.1) \)

We pick a symmetricKey that satisfies the L1Init predicate.

\( 2.3. \text{PICK} \) symmetricKey \( \in \) SymmetricKeyType : L1Init!(symmetricKey)!1

\( \text{BY} \ (2.1) \text{DEF} L1Init \)

We re-state the definitions from the L1Refinement.

\( 2.4. \land \) refPrivateStateEnc \( \in \) PrivateStateEncType

\( \land \) refStateHash \( \in \) HashType

\( \land \) refHistoryStateBinding \( \in \) HashType

\( \text{BY} \ (2.1), (2.2) \text{DEF L1RefinementDefsTypeSafeLemma} \)

We hide the definitions from the L1Refinement.

\( 2.5. \text{HIDE DEF} \) refPrivateStateEnc, refStateHash, refHistoryStateBinding

We re-state the definitions from L1Init.

\( 2.6. \) initialPrivateStateEnc \( \triangleq \) SymmetricEncrypt(L1NVRAM.symmetricKey, HLPrivateState)

\( 2.7. \) initialStateHash \( \triangleq \) Hash(HLPublicState, refPrivateStateEnc)

\( 2.8. \) initialHistoryStateBinding \( \triangleq \) Hash(L1NVRAM.historySummary, refStateHash)

We prove that the definitions from the L1Refinement satisfy their types, using the L1RefinementDefsTypeSafeLemma.

\( 2.9. \land \) initialPrivateStateEnc \( \in \) PrivateStateEncType

\( \land \) initialStateHash \( \in \) HashType

\( \land \) initialHistoryStateBinding \( \in \) HashType

\( \text{BY} \ (2.1), (2.2) \text{DEF L1RefinementDefsTypeSafeLemma} \)

We prove that the definitions from L1Init satisfy their types, using the L1InitDefsTypeSafeLemma.

\( 2.10. \land \) initialPrivateStateEnc \( \in \) PrivateStateEncType

\( \land \) initialStateHash \( \in \) HashType

\( \land \) initialHistoryStateBinding \( \in \) HashType

\( \land \) initialAuthenticator \( \in \) MACType

\( \land \) initialUntrustedStorage \( \in \) L1UntrustedStorageType

\( \land \) initialTrustedStorage \( \in \) L1TrustedStorageType

\( 3.1. \) symmetricKey \( \in \) SymmetricKeyType

\( \text{BY} \ (2.3) \)

\( 3.2. \text{QED} \)

\( \text{BY} \ (2.2), (3.1) \text{DEF L1InitDefsTypeSafeLemma} \)

We hide the definitions from L1Init.

\( 2.11. \text{HIDE DEF} \) initialPrivateStateEnc, initialStateHash, initialHistoryStateBinding,

\( \text{initialAuthenticator}, \text{initialUntrustedStorage}, \text{initialTrustedStorage} \)

A couple of the steps below require knowing that the initial history state binding is authenticated.

\( 2.12. \text{L1HistoryStateBindingAuthenticated}(\text{initialHistoryStateBinding}) \)

The initial authenticator was generated as a MAC of the initial history state binding by L1Init, using a symmetric key that matches the symmetric key in the NVRAM.
We can thus use the MACComplete property to show that the generated MAC validates appropriately. To do this, we first need to prove some types.

For a couple of the steps below, we will need to know that the history state binding in the NVRAM is true.

The initial state hash is bound to the base hash value by the initial history state binding, and the base hash value is the initial value of the history summary in the NVRAM.

The initial history state binding is authenticated.

The bulk of the proof for the initial case is proving that the public and private state defined by the refinement match the initial public and private state.
We show that the refined public state matches the initial public state that the refined encrypted private state matches the encrypted initial state. We will use the \textit{HashCollisionResistant} property.

We begin by proving some types we will need to appeal to the \textit{HashCollisionResistant} property.

We prove that the refined state hash equals the initial state hash. We will use the \textit{UniquenessInvariant}.

Since both history state bindings are authenticated, it follows that the two state hashes are equal.

For the \textit{UniquenessInvariant} to apply, we have to show that the history summary in the initial history state binding is equal to the history summary in the NVRAM.
\(6\). \(LL1NVRAM.\text{historySummary} = \text{BaseHashValue}\)
\(8\). 1. \(LL1NVRAM = [\text{historySummary} \mapsto \text{BaseHashValue},\)
\(\text{symmetricKey} \mapsto \text{symmetricKey}]\)

\(\) By (2)3
\(8\). 2. QED
\(\) By (8)1
\(6\). 3. QED
\(\) By (5)1, (5)2, (5)3, (5)4, (6)1, (6)2
\(\) DEF \text{UniquenessInvariant}, \text{refHistoryStateBinding}, \text{initialHistoryStateBinding}

\(4\). 6. QED
\(\) By (4)1, (4)2, (4)3, (4)4, (4)5, \text{HashCollisionResistant}\) DEF \text{refStateHash}, \text{initialStateHash}

From the previous step, it immediately follows that the refined public state matches the initial public state.

\(3\). 2. \(HL\text{PublicState} = \text{InitialPublicState}\)
\(\) By (3)1

It is slightly more involved to show that the refined private state matches the initial private state. We need to show that the symmetric key that satisfies the \(LL1\text{Init}\) predicate is the same symmetric key that is in the \(NVRAM\), and we also have to employ the correctness of the symmetric crypto operations.

\(3\). 3. \(HL\text{PrivateState} = \text{InitialPrivateState}\)

We need to prove some types.

\(4\). 1. \(LL1NVRAM.\text{symmetricKey} \in \text{SymmetricKeyType}\)
\(\) By (2)2, \(LL1\text{SubtypeImplicationLemma}\) DEF \(LL1\text{SubtypeImplication}\)
\(4\). 2. \(\text{refPrivateStateEnc} = \text{initialPrivateStateEnc}\)
\(\) By (3)1

The refined private state equals the decryption of the refined encrypted private state, using the symmetry key brought into existence in \(LL1\text{Init}\).

\(4\). 3. \(HL\text{PrivateState} = \text{SymmetricDecrypt}(\text{symmetricKey}, \text{refPrivateStateEnc})\)

The refined private state equals the decryption of the refined encrypted private state, using the symmetry key in the \(NVRAM\). This follows because the refined encrypted private state is defined in the \(LL1\text{Refinement}\) as the encryption of the refined private state, and the symmetric crypto operations are assumed to be correct.

\(5\). 1. \(HL\text{PrivateState} = \text{SymmetricDecrypt}(LL1NVRAM.\text{symmetricKey}, \text{refPrivateStateEnc})\)
\(\) By (2)1, (2)7 DEF \(LL1\text{Refinement}\)
\(6\). 2. QED
\(\) By (4)1, (6)1, \text{SymmetricCryptoCorrect}\) DEF \text{refPrivateStateEnc}

The symmetric key in the \(NVRAM\) matches the symmetric key brought into existence in \(LL1\text{Init}\).

\(5\). 2. \(LL1NVRAM.\text{symmetricKey} = \text{symmetricKey}\)
\(\) By (2)3
\(6\). 2. QED
\(\) By (6)1
\(5\). 3. QED
\(\) By (5)1, (5)2

The initial private state equals the decryption of the initial encrypted private state, using the symmetry key brought into existence in \(LL1\text{Init}\). This follows because the refined encrypted private state is defined in \(LL1\text{Init}\) as the encryption of the initial private state, and the symmetric crypto operations are assumed to be correct.

\(4\). 4. \(\text{InitialPrivateState} = \text{SymmetricDecrypt}(\text{symmetricKey}, \text{initialPrivateStateEnc})\)
\(5\). 1. \(\text{InitialPrivateState} \in \text{PrivateStateType}\)
\(\) By \text{ConstantsTypeSafe}\)
\(5\). 2. QED
\(\) By (4)1, (5)1, \text{SymmetricCryptoCorrect}\) DEF \text{initialPrivateStateEnc}
(4)5. QED
   BY (4)2, (4)3, (4)4

The truth of the conjuncts implies the truth of the conjunction.

(3)4. QED
   BY (3)2, (3)3

The QED step simply asserts each conjunct in the $HL_{init}$ predicate.

(2)9. QED
(3)1. $HL_{Alive} = TRUE$
   (4)1. $LL1_{NVRAM\text{HistorySummaryUncorrupted}}$
      BY (2)7
   (4)2. QED
      BY (2)1, (4)1 DEF $LL1_{Refinement}$
(3)2. $HL_{AvailableInputs} = InitialAvailableInputs$
   (4)1. $LL1_{AvailableInputs} = InitialAvailableInputs$
      BY (2)3
   (4)2. $HL_{AvailableInputs} = LL1_{AvailableInputs}$
      BY (2)1 DEF $LL1_{Refinement}$
(4)3. QED
   BY (4)1, (4)2
(3)3. $HL_{ObservedOutputs} = \{\}$
   (4)1. $LL1_{ObservedOutputs} = \{\}$
      BY (2)3
   (4)2. $HL_{ObservedOutputs} = LL1_{ObservedOutputs}$
      BY (2)1 DEF $LL1_{Refinement}$
(4)3. QED
   BY (4)1, (4)2
(3)4. $HL_{PublicState} = InitialPublicState$
   BY (2)8
(3)5. $HL_{PrivateState} = InitialPrivateState$
   BY (2)8
(3)6. QED
   BY (3)1, (3)2, (3)3, (3)4, (3)5 DEF $HL_{init}$

For the induction step, we will need the refinement, the type invariant, and the correctness invariants to be true in both the unprimed and primed states.

(1)4. ($\land [LL1_{Next}]_{LL1\_Vars}$
   $\land LL1\_Refinement$
   $\land LL1\_Refinement'$
   $\land LL1\_TypeInvariant$
   $\land LL1\_TypeInvariant'$
   $\land CorrectnessInvariants$
   $\land CorrectnessInvariants'$)
   $\Rightarrow$
   $[HL_{Next}]_{HL\_Vars}$

We assume the antecedents.

(2)1. HAVE
   $\land [LL1_{Next}]_{LL1\_Vars}$
   $\land LL1\_Refinement$
   $\land LL1\_Refinement'$
   $\land LL1\_TypeInvariant$
   $\land LL1\_TypeInvariant'$
   $\land CorrectnessInvariants$
   $\land CorrectnessInvariants'$

167
We then prove that each step in the Memoir-Basic spec refines to a step in the high-level spec. First, a Memoir-Basic stuttering step refines to a high-level stuttering step.

(2) \text{UNCHANGED } LL1\text{Vars} \Rightarrow \text{UNCHANGED } HL\text{Vars}

(3) \text{HAVE UNCHANGED } LL1\text{Vars}

The \textit{HLAlive} predicate is unchanged because the \textit{LL1NVRAMHistorySummaryUncorrupted} predicate is unchanged.

(3) \text{UNCHANGED } HL\text{Alive}

The \textit{LL1NVRAMHistorySummaryUncorruptedEqualsHLAliveLemma} tells us that the \textit{LL1NVRAMHistorySummaryUncorrupted} and \textit{HLAlive} are equal.

(4) \text{UNCHANGED } LL1\text{NVRAMHistorySummaryUncorrupted}

(5) \text{UNCHANGED } LL1\text{NVRAM}

The mapping from \textit{LL1AvailableInputs} to \textit{HLAvailableInputs} is direct.

(3) \text{UNCHANGED } HL\text{AvailableInputs}

The mapping from \textit{LL1ObservedOutputs} to \textit{HLObservedOutputs} is direct.

(3) \text{UNCHANGED } HLObservedOutputs
We prove the stuttering of the high-level public and private state by using the *NonAdvancementLemma*. Many of the antecedents for the *NonAdvancementLemma* come directly from antecedents in the induction.

The *LL1NVRAM* is unchanged because the Memoir-Basic variables are unchanged.

The *UnchangedAuthenticatedHistoryStateBindingsLemma* tells us that there is no change to the set of history state bindings that have authenticators in the set *LL1ObservedAuthenticators*.

We have all of the antecedents for the *NonAdvancementLemma*, so we can apply it directly.

A Memoir-Basic *LL1MakeInputAvailable* action refines to a high-level *HLMakeInputAvailable* action.

The mapping from *LL1AvailableInputs* to *HLAvailableInputs* is direct.

The mapping from *LL1ObservedOutputs* to *HLObservedOutputs* is direct.
The **HLAlive** predicate is unchanged because the **LL1NVRAMHistorySummaryUncorrupted** predicate is unchanged.

The **LL1NVRAMHistorySummaryUncorruptedEqualsHLAliveLemma** tells us that **LL1NVRAMHistorySummaryUncorrupted** and **HLAlive** are equal.

Then, the **LL1NVRAMHistorySummaryUncorruptedUnchangedLemma** tells us that the **LL1NVRAMHistorySummaryUncorrupted** predicate is unchanged.

We prove the stuttering of the high-level public and private state by using the **NonAdvancementLemma**.

Many of the antecedents for the **NonAdvancementLemma** come directly from antecedents in the induction.

The **NVRAM** is unchanged by definition of the **LL1MakeInputAvailable** action.

We prove the stuttering of the high-level public and private state by using the **NonAdvancementLemma**.
The *UnchangedAuthenticatedHistoryStateBindingsLemma* tells us that there is no change to the set of history state bindings that have authenticators in the set \( \text{LL1ObservedAuthenticators} \).

(4) 3. \( \forall \text{historyStateBinding} \in \text{HashType} : \)

\[
\text{UNCHANGED LLIHistoryStateBindingAuthenticated}(\text{historyStateBinding})
\]

(5) 1. \( \text{UNCHANGED } \langle \text{LL1NVRAM, LL1ObservedAuthenticators} \rangle \)

BY (3) 2

(5) 2. QED

BY (5) 1, *UnchangedAuthenticatedHistoryStateBindingsLemma*

We have all of the antecedents for the *NonAdvancementLemma*, so we can apply it directly.

(4) 4. QED

BY (4) 1, (4) 2, (4) 3, *NonAdvancementLemma*

(3) 8. QED

BY (3) 2, (3) 3, (3) 4, (3) 5, (3) 6, (3) 7 DEF *HLMakeInputAvailable*

A Memoir-Basic \( \text{LL1PerformOperation} \) action refines to a high-level \( \text{HLAdvanceService} \) action.

(2) 4. \( \text{LL1PerformOperation} \Rightarrow \text{HLAdvanceService} \)

We assume the antecedent.

(3) 1. HAVE \( \text{LL1PerformOperation} \)

We pick an input that satisfies the \( \text{LL1PerformOperation} \) predicate.

(3) 2. PICK \( \text{input} \in \text{LL1AvailableInputs} : \text{LL1PerformOperation!}(\text{input})!1 \)

BY (3) 1 DEF \( \text{LL1PerformOperation} \)

We re-state the definitions from the \( \text{LL1Refinement} \)

(3) \( \text{refPrivateStateEnc} \triangleq \text{SymmetricEncrypt}(\text{LL1NVRAM.symmetricKey, HLPrivateState}) \)

(3) \( \text{refStateHash} \triangleq \text{Hash}(\text{HLPublicState, refPrivateStateEnc}) \)

(3) \( \text{refHistoryStateBinding} \triangleq \text{Hash}(\text{LL1NVRAM.historySummary, refStateHash}) \)

We prove that the definitions from the \( \text{LL1Refinement} \) satisfy their types, using the \( \text{LL1RefinementDefsTypeSafeLemma} \).

(3) 3. \( \forall \text{refPrivateStateEnc} \in \text{PrivateStateEncType} \)

\( \land \text{refStateHash} \in \text{HashType} \)

\( \land \text{refHistoryStateBinding} \in \text{HashType} \)

(4) 1. \( \text{LL1Refinement} \)

BY (2) 1

(4) 2. \( \text{LL1TypeInvariant} \)

BY (2) 1

(4) 3. QED

BY (4) 1, (4) 2, \( \text{LL1RefinementDefsTypeSafeLemma} \)

We also prove that the primed definitions from the \( \text{LL1Refinement} \) satisfy their types, using the \( \text{LL1RefinementPrimeDefsTypeSafeLemma} \).

(3) 4. \( \forall \text{refPrivateStateEnc}' \in \text{PrivateStateEncType} \)

\( \land \text{refStateHash}' \in \text{HashType} \)

\( \land \text{refHistoryStateBinding}' \in \text{HashType} \)

(4) 1. \( \text{LL1Refinement}' \)

BY (2) 1

(4) 2. \( \text{LL1TypeInvariant}' \)

BY (2) 1

(4) 3. QED

BY (4) 1, (4) 2, \( \text{LL1RefinementPrimeDefsTypeSafeLemma} \)

We hide the definitions from the \( \text{LL1Refinement} \).

(3) HIDE DEF \( \text{refPrivateStateEnc, refStateHash, refHistoryStateBinding} \)

We re-state the definitions from \( \text{LL1PerformOperation} \).

(3) \( \text{stateHash} \triangleq \text{Hash}(\text{LL1RAM.publicState, LL1RAM.privateStateEnc}) \)
We prove that the history summary in the NVRAM is uncorrupted in the unprimed state. This follows from the enablement conditions of the LL1PerformOperation action. We will expand the definitions of LL1NVRAMHistorySummaryUncorrupted and LL1HistoryStateBindingAuthenticated, and prove that the required conditions are all satisfied.

(3.6) LL1NVRAMHistorySummaryUncorrupted
The authenticator in the RAM authenticates the history state binding, since this is an enablement condition of the LL1PerformOperation action.

(4.1) ValidateMAC(LL1NVRAM.symmetricKey, historyStateBinding, LL1RAM.authenticator)
by (3) 2 def historyStateBinding, stateHash
The state hash defined in the LL1PerformOperation is in HashType.

(4.2) stateHash ∈ HashType
by (3) 5
From the UnforgeabilityInvariant, the authenticator in the RAM is in the set of observed authenticators.

(4.3) LL1RAM.authenticator ∈ LL1ObservedAuthenticators
(5) 1. UnforgeabilityInvariant
BY (2)1 DEF CorrectnessInvariants
(5)2. historyStateBinding ∈ HashType
BY (3)5
(5)3. QED
BY (4)1, (5)1, (5)2 DEF UnforgeabilityInvariant

The above three conditions are sufficient to satisfy the \textit{LL1NVRAMHistorySummaryUncorrupted} predicate in the unprimed state, given that the history summary in the RAM equals the history summary in the NVRAM.

(4)4. LL1NVRAM.historySummary = LL1RAM.historySummary
BY (3)2
(4)5. QED
BY (4)1, (4)2, (4)3, (4)4
DEF LL1NVRAMHistorySummaryUncorrupted, LL1HistoryStateBindingAuthenticated, historyStateBinding

We prove that the history summary in the NVRAM is uncorrupted in the primed state. This follows from the enablement conditions of the \textit{LL1PerformOperation} action. We will expand the definitions of \textit{LL1NVRAMHistorySummaryUncorrupted} and \textit{LL1HistoryStateBindingAuthenticated}, and prove that the required conditions are all satisfied.

(3)7. LL1NVRAMHistorySummaryUncorrupted'

The new authenticator defined by the \textit{LL1PerformOperation} action authenticates the new history state binding defined by this action. We will use the MACComplete property.

(4)1. ValidateMAC(LL1NVRAM.symmetricKey', newHistoryStateBinding, newAuthenticator)
The new authenticator was generated as a MAC of the new history state binding by \textit{LL1PerformOperation}, using the unchanged symmetric key in the NVRAM.

(5)1. newAuthenticator = GenerateMAC(LL1NVRAM.symmetricKey', newHistoryStateBinding)
(6)1. UNCHANGED LL1NVRAM.symmetricKey
BY (2)1, SymmetricKeyConstantLemma
(6)2. QED
BY (3)2, (6)1
DEF newAuthenticator, newHistoryStateBinding, newStateHash, newPrivateStateEnc, sResult, privateState

We can thus use the MACComplete property to show that the generated MAC validates appropriately. To do this, we first need to prove some types.

(5)2. LL1NVRAM.symmetricKey' ∈ SymmetricKeyType
(6)1. LL1TypeInvariant'
BY (2)1
(6)2. QED
BY (6)1, LL1SubtypeImplicationLemma DEF LL1SubtypeImplication
(5)3. newHistoryStateBinding ∈ HashType
BY (3)5

Then, we appeal to the MACComplete property in a straightforward way.

(5)4. QED
BY (5)1, (5)2, (5)3, MACComplete

The new state hash defined by the \textit{LL1PerformOperation} is in HashType.

(4)2. newStateHash ∈ HashType
BY (3)5

The new authenticator defined by the \textit{LL1PerformOperation} action is in the primed set of observed authenticators, as specified by the \textit{LL1PerformOperation} action.

(4)3. newAuthenticator ∈ LL1ObservedAuthenticators'
(5)1. LL1ObservedAuthenticators' = LL1ObservedAuthenticators ∪ \{newAuthenticator\}
BY (3)2
DEF newAuthenticator, newHistoryStateBinding, newStateHash,
newHistorySummary, newPrivateStateEnc, sResult, privateState

(5) 2. QED
BY (5) 1

The history summary in the primed state of the NVRAM equals the new history summary defined by the LL1PerformOperation.

(4) 4. LL1NVRAM.historySummary' = newHistorySummary
(5) 1. LL1NVRAM' = [
    historySummary ↦ newHistorySummary,
    symmetricKey ↦ LL1NVRAM.symmetricKey]
BY (3) 2 DEF newHistorySummary
(5) 2. QED
BY (5) 1

The above three conditions and the equality are sufficient to satisfy the LL1NVRAMHistorySummaryUncorrupted predicate in the primed state.

(4) 5. QED
BY (4) 1, (4) 2, (4) 3, (4) 4
DEF LL1NVRAMHistorySummaryUncorrupted, LL1HistoryStateBindingAuthenticated,
newHistoryStateBinding

The proof proper has two main steps. The first main step is to prove that the public and private states that are arguments to the Service in LL1PerformOperation are identical to the values of HLPublicState and HLPriavteState in the LL1Refinement.

(3) 8. ∧ HLPublicState = LL1RAM.publicState
    ∧ HLPriavteState = privateState

We show that the refined public state matches the public state in the RAM and that the refined encrypted private state matches the encrypted private state in the RAM. We will use the HashCollisionResistant property.

(4) 1. ∧ HLPublicState = LL1RAM.publicState
    ∧ refPrivateStateEnc = LL1RAM.privateStateEnc

We begin by proving some types we will need to appeal to the HashCollisionResistant property.

(5) 1. HLPublicState ∈ HashDomain
    (6) 1. LL1RAM.publicState ∈ HashDomain
        ∧ LL1RAM.privateStateEnc ∈ HashDomain
    BY (2) 1, (3) 6 DEF LL1Refinement
    (6) 2. QED
    BY (6) 1 DEF HashDomain
(5) 2. ∧ LL1RAM.publicState ∈ HashDomain
    ∧ LL1RAM.privateStateEnc ∈ HashDomain
    (6) 1. ∧ LL1RAM.publicState ∈ PublicStateType
        ∧ LL1RAM.privateStateEnc ∈ PrivateStateEncType
(7) 1. LL1TypeInvariant
    BY (2) 1
    (7) 2. QED
    BY (7) 1, LL1SubtypeImplicationLemmaDEF LL1SubtypeImplication
    (6) 2. QED
    BY (6) 1 DEF HashDomain
(5) 3. refPrivateStateEnc ∈ HashDomain
    (6) 1. refPrivateStateEnc ∈ PrivateStateEncType
        BY (3) 3
    (6) 2. QED
    BY (6) 1 DEF HashDomain

We then prove that the refined state hash equals the state hash defined by LL1PerformOperation. We will use the UniquenessInvariant.
(5) 4. \( \text{refStateHash} = \text{stateHash} \)

We prove some types we will need for the UniquenessInvariant.

(6) 1. \( \text{refStateHash} \in \text{HashType} \)
   BY (3)/3
(6) 2. \( \text{stateHash} \in \text{HashType} \)
   BY (3)/5

We prove that the refined history state binding is authenticated. This follows directly from the LL1Refinement.

(6) 3. \( \text{LL1HistoryStateBindingAuthenticated} (\text{refHistoryStateBinding}) \)
   BY (2)/1, (3)/6 DEF LL1Refinement, refHistoryStateBinding, refStateHash, refPrivateStateEnc

We prove that the history state binding defined by the LL1PerformOperation action is authenticated.

(6) 4. \( \text{LL1HistoryStateBindingAuthenticated} (\text{historyStateBinding}) \)
   The authenticator in the RAM authenticates the history state binding, since this is an enablement condition of the LL1PerformOperation action.

(7) 1. \( \text{ValidateMAC} (\text{LL1NVRAM} . \text{symmetricKey}, \text{historyStateBinding}, \text{LL1RAM} . \text{authenticator}) \)
   BY (3)/2 DEF historyStateBinding, stateHash

From the UnforgeabilityInvariant, the authenticator in the RAM is in the set of observed authenticators.

(7) 2. \( \text{LL1RAM} . \text{authenticator} \in \text{LL1ObservedAuthenticators} \)
   (8) 1. UnforgeabilityInvariant
       BY (2)/1 DEF CorrectnessInvariants
(8) 2. \( \text{historyStateBinding} \in \text{HashType} \)
       BY (3)/5
(8) 3. QED
       BY (7)/1, (8)/1, (8)/2 DEF UnforgeabilityInvariant

The above two conditions satisfy the definition of LL1HistoryStateBindingAuthenticated.

(7) 3. QED
   BY (7)/1, (7)/2 DEF LL1HistoryStateBindingAuthenticated

Since both history state bindings are authenticated, it follows that the two state hashes are equal.

(6) 5. QED
(7) 1. UniquenessInvariant
    BY (2)/1 DEF CorrectnessInvariants
(7) 2. \( \text{LL1NVRAM} . \text{historySummary} = \text{LL1RAM} . \text{historySummary} \)
    BY (3)/2
(7) 3. QED
    BY (7)/1, (7)/2, (6)/1, (6)/2, (6)/3, (6)/4
    DEF UniquenessInvariant, refHistoryStateBinding, historyStateBinding
(5) 5. QED
    BY (5)/1, (5)/2, (5)/3, (5)/4, HashCollisionResistant DEF refStateHash, stateHash

From the previous step, it immediately follows that the refined public state matches the public state in the RAM.

(4) 2. \( \text{HLPublicState} = \text{LL1RAM} . \text{publicState} \)
   BY (4)/1

It is slightly more involved to show that the refined private state matches the private state that is the decryption of the encrypted private state in the RAM. We need to employ the correctness of the symmetric crypto operations.

(4) 3. \( \text{HLPPrivateState} = \text{privateState} \)
(5) 1. \( \text{refPrivateStateEnc} = \text{LL1RAM} . \text{privateStateEnc} \)
    BY (4)/1
(5) 2. \( \text{LL1NVRAM} . \text{symmetricKey} \in \text{SymmetricKeyType} \)
(6) 1. LL1TypeInvariant
    BY (2)/1
(6) 2. QED

175
The second main step is to prove that the public and private states that are produced by the Service in $LL_1$PerformOperation are identical to the primed values of $HL_{PublicState}$ and $HL_{PrivateState}$ in the $LL_1$Refinement.

We show that the primed refined public state matches the public state produced by the invocation of Service in $LL_1$PerformOperation, and that the primed refined encrypted private state matches the encryption of the private state produced by the invocation of Service in $LL_1$PerformOperation. We will use the HashCollisionResistant property.

We prove some types we will need for the HashCollisionResistant property.

We then prove that the primed refined state hash equals the new state hash defined by $LL_1$PerformOperation. We will use the UniquenessInvariant.

We prove that the refined history state binding is authenticated in the primed state. This follows directly from the $LL_1$Refinement.

Ideally, at this point we would prove that the new history state binding defined by the $LL_1$PerformOperation action is authenticated in the primed state. However, we cannot prove this, because TLA+ does not allow us to prime an operator, only an expression. If we prime the entire expression, the newHistoryStateBinding argument also becomes primed, which changes its meaning. So instead, we merely prove that the new authenticator defined by the $LL_1$PerformOperation action authenticates the new history state binding. We will use the MACComplete property.
(6) ValidateMAC(LL1NVRAM.symmetricKey', newHistoryStateBinding, newAuthenticator)

The new authenticator was generated as a MAC of the new history state binding by LL1PerformOperation, using the unchanged symmetric key in the NVRAM.

(7) newAuthenticator = GenerateMAC(LL1NVRAM.symmetricKey', newHistoryStateBinding)

We can thus use the MACComplete property to show that the generated MAC validates appropriately. To do this, we first need to prove some types.

(8) newHistoryStateBinding ∈ HashType

Then, we appeal to the MACComplete property in a straightforward way.

Ideally, at this point we would prove that because both history state bindings are authenticated, it follows that the two state hashes are equal. However, we have not proven that the new state binding defined by the LL1PerformOperation action is authenticated in the primed state, because TLA+ does not allow us to prime an operator, only an expression. So, we have to expand the definition of LL1HistoryStateBindingAuthenticated for the steps of this proof.

(9) This is the definition of the UniquenessInvariant, with the LETs instantiated and LL1HistoryStateBindingAuthenticated fully expanded.

(7) ∀ stateHash1, stateHash2 ∈ HashType :

(8) UniquenessInvariant'

The universal quantifiers in the previous step range over HashType, so we need to prove that the state hashes are in HashType.
(7)3. $\text{newStateHash} \in \text{HashType}$  
BY (3)5
This is the first conjunct in the antecedent of the implication in the expanded UniquenessInvariant. It follows directly from the fact that the refined history state binding is authenticated in the primed state.

(7)4. $\exists \text{authenticator} \in \text{LL1ObservedAuthenticators'}: $  
\begin{align*}
\text{ValidateMAC(} & \\
& \text{LL1NVRAM}.\text{symmetricKey'}, \\
& \text{Hash}\text{(LL1NVRAM}.\text{historySummary'}, \text{refStateHash'}), \\
& \text{authenticator})
\end{align*}
BY (6)1 DEF LL1HistoryStateBindingAuthenticated, refHistoryStateBinding
This is the second conjunct in the antecedent of the implication in the expanded UniquenessInvariant. The new authenticator defined by the LL1PerformOperation action will serve as a witness for the existential quantifier.

(7)5. $\exists \text{authenticator} \in \text{LL1ObservedAuthenticators'}: $  
\begin{align*}
\text{ValidateMAC(} & \\
& \text{LL1NVRAM}.\text{symmetricKey'}, \\
& \text{Hash}\text{(LL1NVRAM}.\text{historySummary'}, \text{newStateHash}), \\
& \text{authenticator})
\end{align*}
The new authenticator defined by the LL1PerformOperation action is in the primed set of observed authenticators, as specified by the LL1PerformOperation action.

(8)1. $\text{newAuthenticator} \in \text{LL1ObservedAuthenticators'}$
(9)1. $\text{LL1ObservedAuthenticators'} = \text{LL1ObservedAuthenticators} \cup \{\text{newAuthenticator}\}$  
BY (3)2
DEF newAuthenticator, newHistoryStateBinding, newStateHash, newHistorySummary, newPrivateStateEnc, sResult, privateState
(9)2. QED  
BY (9)1
The new authenticator defined by the LL1PerformOperation action authenticates the new history state binding defined by this action. Because the history summary in the primed NVRAM equals the new history summary defined by the action, we can derive an expression that satisfies the existential expression above.

(8)2. ValidateMAC(  
\begin{align*}
& \text{LL1NVRAM}.\text{symmetricKey'}, \\
& \text{Hash}\text{(LL1NVRAM}.\text{historySummary'}, \text{newStateHash}), \\
& \text{newAuthenticator})
\end{align*}
(9)1. $\text{newHistoryStateBinding} = \text{Hash}(\text{LL1NVRAM}.\text{historySummary'}, \text{newStateHash})$
(10)1. $\text{LL1NVRAM}.\text{historySummary'} = \text{newHistorySummary}$  
(11)1. $\text{LL1NVRAM'} = [ $  
\begin{align*}
& \text{historySummary} \mapsto \text{newHistorySummary}, \\
& \text{symmetricKey} \mapsto \text{LL1NVRAM}.\text{symmetricKey}]
\end{align*}$  
BY (3)2 DEF newHistorySummary  
(11)2. QED  
BY (11)1  
(10)2. QED  
BY (10)1 DEF newHistoryStateBinding  
(9)2. QED  
BY (6)2, (9)1  
(8)3. QED  
BY (8)1, (8)2
With both conjuncts in the antecedent true, the conclusion readily follows.

(7)6. QED
(5)6. QED

Ideally, this QED step should just read:

BY (5)1, (5)2, (5)3, (5)4, (5)5, \texttt{HashCollisionResistant}

However, the prover seems to get a little confused in this instance. We make life easier for the prover by defining some local variables and hiding their definitions before appealing to the \texttt{HashCollisionResistant} assumption.

(6) \( h_1 a \overset{\Delta}{=} \text{HLPrivateState}' \)
(6) \( h_2 a \overset{\Delta}{=} \text{sResult}.newPublicState' \)
(6) \( h_1 b \overset{\Delta}{=} \text{sResult}.newPublicState \)
(6) \( h_2 b \overset{\Delta}{=} \text{newPrivateStateEnc} \)
(6)1. \( h_1 a \in \text{HashDomain} \)
BY (5)1
(6)2. \( h_2 a \in \text{HashDomain} \)
BY (5)2
(6)3. \( h_1 b \in \text{HashDomain} \)
BY (5)3
(6)4. \( h_2 b \in \text{HashDomain} \)
BY (5)4
(6)5. \( \text{Hash}(h_1 a, h_2 a) = \text{Hash}(h_1 b, h_2 b) \)
BY (5)5 DEF \text{refStateHash}, \text{newStateHash}
(6)6. \( h_1 a = h_1 b \land h_2 a = h_2 b \)
(7) HIDE DEF \( h_1 a, h_2 a, h_1 b, h_2 b \)
(7)1. QED
BY (6)1, (6)2, (6)3, (6)4, (6)5, \texttt{HashCollisionResistant}
(6)7. QED
BY (6)6

To show that the primed refined high-level private state matches the private state that is produced by the Service invocation in \texttt{LL1PerformOperation}, we need to appeal to the correctness of the symmetric crypto operations and the \texttt{SymmetricKeyConstantLemma}.

(4)2. \( \text{HLPrivateState}' = \text{sResult}.newPrivateState \)
(5)1. \texttt{SymmetricDecrypt}(\text{LL1NVRAM}.symmetricKey', \text{refPrivateStateEnc'}) =
\texttt{SymmetricDecrypt}(\text{LL1NVRAM}.symmetricKey, \text{newPrivateStateEnc})
(6)1. \text{refPrivateStateEnc'} = \text{newPrivateStateEnc}
BY (4)1
(6)2. \text{LL1NVRAM}.symmetricKey' = \text{LL1NVRAM}.symmetricKey
BY (2)1, \texttt{SymmetricKeyConstantLemma}
(6)20. QED
BY (6)1, (6)2
(5)2. \( \text{HLPrivateState}' = \text{SymmetricDecrypt}(\text{LL1NVRAM}.symmetricKey', \text{refPrivateStateEnc'}) \)
(6)1. \text{LL1NVRAM}.symmetricKey' \in \text{SymmetricKeyType}
(7)1. \text{LL1TypeInvariant}'
BY (2)1
(7)2. QED
BY (7)1, \texttt{LL1SubtypeImplicationLemma} DEF \text{LL1SubtypeImplication}
(6)2. \text{HLPrivateState}' \in \text{PrivateStateType}
BY (2)1, (3)7 DEF \text{LL1Refinement}
(6)3. QED
BY (6)1, (6)2, \texttt{SymmetricCryptoCorrect} DEF \text{refPrivateStateEnc}
(5)3. \( \text{sResult}.\text{newPrivateState} = \text{SymmetricDecrypt}(\text{LL1NVRAM}.symmetricKey, \text{newPrivateStateEnc}) \)
(6)1. \text{LL1NVRAM}.symmetricKey \in \text{SymmetricKeyType}
(7)1. **LL1TypeInvariant**  
   BY (2)1
(7)2. QED  
   BY (7)1, **LL1SubtypeImplicationLemma**Def **LL1SubtypeImplication**

(6)2. `sResult.newPrivateState` ∈ **PrivateState**Type  
   BY (3)5
(6)3. QED  
   BY (6)1, (6)2, **SymmetricCryptoCorrect**Def `newPrivateStateEnc`

(5)4. QED  
   BY (5)1, (5)2, (5)3
(4)3. QED  
   BY (4)1, (4)2

The QED step states each conjunct of the **HLAdvanceService** action, which all follow directly from the two main steps above.

(3)10. QED  
   (4)1. `input` ∈ **HLAvailableInputs**  
      (5)1. **HLAvailableInputs** = **LL1AvailableInputs**  
         BY (2)1 DEF **LL1Refinement**
(5)2. QED  
         BY (5)1, (3)2
(4)2. **HLAlive** = true  
   BY (2)1, (3)6 DEF **LL1Refinement**
(4)3. **HLPublicState′** = `hlSResult.newPublicState`  
   (5)1. **HLPublicState′** = `sResult.newPublicState`  
      BY (3)9
(5)2. `sResult.newPublicState` = `hlSResult.newPublicState`  
   (6)1. `sResult` = `hlSResult`  
      BY (3)8 DEF `sResult`, `hlSResult`
(6)2. QED  
      BY (6)1
(5)3. QED  
      BY (5)1, (5)2
(4)4. **HLPrivateState′** = `hlSResult.newPrivateState`  
   (5)1. **HLPrivateState′** = `sResult.newPrivateState`  
      BY (3)9
(5)2. `sResult.newPrivateState` = `hlSResult.newPrivateState`  
   (6)1. `sResult` = `hlSResult`  
      BY (3)8 DEF `sResult`, `hlSResult`
(6)2. QED  
      BY (6)1
(5)3. QED  
      BY (5)1, (5)2
(4)5. **HLObservedOutputs′** = **HLObservedOutputs** ∪ `{`hlSResult.output`}`  
   (5)1. **LL1ObservedOutputs′** = **LL1ObservedOutputs** ∪ `{`sResult.output`}`  
      BY (3)2 DEF `sResult`, `privateState`
(5)2. **HLObservedOutputs** = **LL1ObservedOutputs**  
   BY (2)1 DEF **LL1Refinement**
(5)3. **HLObservedOutputs′** = **LL1ObservedOutputs′**  
   BY (2)1 DEF **LL1Refinement**
(5)4. `sResult.output` = `hlSResult.output`  
   (6)1. `sResult` = `hlSResult`  
      BY (3)8 DEF `sResult`, `hlSResult`
A Memoir-Basic $LL1\text{RepeatOperation}$ action refines to a high-level stuttering step.

(2.5). $LL1\text{RepeatOperation} \Rightarrow \text{UNCHANGED } HLVars$

(3.1). HAVE $LL1\text{RepeatOperation}$

(3.2). PICK input $\in LL1\text{AvailableInputs}$ : $LL1\text{RepeatOperation}!(input)!1$

Then, the $LL1\text{NVRAMHistorySummaryUncorrupted}$ predicate is unchanged because the $LL1\text{NVRAMHistorySummaryUncorrupted}$ predicate is unchanged.

(3.3). UNCHANGED $HLAlive$

The $LL1\text{NVRAMHistorySummaryUncorrupted} = HLAlive$ tells us that $LL1\text{NVRAMHistorySummaryUncorrupted}$ and $HLAlive$ are equal.

(4.1). $LL1\text{NVRAMHistorySummaryUncorrupted} = HLAlive$

(5.1). $LL1\text{Refinement}$

(5.2). QED

By (5.1), $LL1\text{NVRAMHistorySummaryUncorrupted} = HLAlive$

(4.2). $LL1\text{NVRAMHistorySummaryUncorrupted}' = HLAlive'$

(5.1). $LL1\text{Refinement}'$

(5.2). QED

Then, the $LL1\text{NVRAMHistorySummaryUncorrupted} = HLAlive$ tells us that the $LL1\text{NVRAMHistorySummaryUncorrupted}$ predicate is unchanged.

(4.3). UNCHANGED $LL1\text{NVRAMHistorySummaryUncorrupted}$

(5.1). $LL1\text{TypeInvariant}$

(5.2). QED

The $NVRAM$ is unchanged by definition of the $LL1\text{RepeatOperation}$ action.

(5.2). UNCHANGED $LL1\text{NVRAM}$

By (3.2)

The $LL1\text{RepeatOperation}UnchangedAuthenticatedHistoryStateBindingsLemma$ tells us that there is no change to the set of history state bindings that have authenticators in the set $LL1\text{ObservedAuthenticators}$.

(5.3). $\forall \text{historyStateBinding} \in HashType :$

UNCHANGED $LL1\text{HistoryStateBindingAuthenticated}(\text{historyStateBinding})$

(6.1). $LL1\text{TypeInvariant} \land \text{UnforgeabilityInvariant} \land \text{InclusionInvariant}$

By (2.1) DEF CorrectnessInvariants
The mapping from $\text{LL1AvailableInputs}$ to $\text{HLAvailableInputs}$ is direct.

(3.4) UNCHANGED $\text{HLAvailableInputs}$

(4.1) UNCHANGED $\text{LL1AvailableInputs}$

(4.2) $\wedge \text{HLAvailableInputs} = \text{LL1AvailableInputs}$

(4.3) QED

The mapping from $\text{LL1ObservedOutputs}$ to $\text{HLObservedOutputs}$ is direct.

(3.5) UNCHANGED $\text{HLObservedOutputs}$

(4.1) UNCHANGED $\text{LL1ObservedOutputs}$

(5.1) $\text{LL1TypeInvariant} \land \text{UnforgeabilityInvariant} \land \text{InclusionInvariant}$

(5.2) QED

We prove the stuttering of the high-level public and private state by using the $\text{NonAdvancementLemma}$.  

(3.6) UNCHANGED $\langle \text{HLPublicState}, \text{HLPrivateState} \rangle$

Many of the antecedents for the $\text{NonAdvancementLemma}$ come directly from antecedents in the induction.

(4.1) $\wedge \text{LL1Refinement}$

(4.2) $\wedge \text{LL1TypeInvariant}$

(4.3) QED

The $\text{NVRAM}$ is unchanged by definition of the $\text{LL1RepeatOperation}$ action.

(4.2) UNCHANGED $\text{LL1NVRAM}$

The $\text{LL1RepeatOperationUnchangedAuthenticatedHistoryStateBindingsLemma}$ tells us that there is no change to the set of history state bindings that have authenticators in the set $\text{LL1ObservedAuthenticators}$.

(4.3) $\forall \text{historyStateBinding} \in \text{HashType}$ :

(5.1) $\text{LL1HistoryStateBindingAuthenticated(historyStateBinding)}$

(5.2) QED
A Memoir-Basic $LL1_{\text{Restart}}$ action refines to a high-level stuttering step.

(2). $LL1_{\text{Restart}} \Rightarrow \text{UNCHANGED HLVars}$

(3).1. HAVE $LL1_{\text{Restart}}$

(3).2. PICK $\text{untrustedStorage} \in LL1_{\text{UntrustedStorageType}}$, $\text{randomSymmetricKey} \in \text{SymmetricKeyType} \setminus \{ LL1_{\text{NVRAM}}.\text{symmetricKey} \}$, $\text{hash} \in \text{HashType}$:

$LL1_{\text{Restart}}!(\text{untrustedStorage}, \text{randomSymmetricKey}, \text{hash})$

BY (3).1 DEF $LL1_{\text{Restart}}$

The $HL_{\text{Alive}}$ predicate is unchanged because the $LL1_{\text{NVRAMHistorySummaryUncorrupted}}$ predicate is unchanged.

(3).3. UNCHANGED $HL_{\text{Alive}}$

The $LL1_{\text{NVRAMHistorySummaryUncorruptedEqualsHLAliveLemma}}$ tells us that $LL1_{\text{NVRAMHistorySummaryUncorrupted}}$ and $HL_{\text{Alive}}$ are equal.

(4).1. $LL1_{\text{NVRAMHistorySummaryUncorrupted}} = HL_{\text{Alive}}$

(5).1. $LL1_{\text{Refinement}}$

BY (2).1

(5).2. QED

BY (5).1, $LL1_{\text{NVRAMHistorySummaryUncorruptedEqualsHLAliveLemma}}$

(4).2. $LL1_{\text{NVRAMHistorySummaryUncorrupted}}' = HL_{\text{Alive}}'$

(5).1. $LL1_{\text{Refinement}}'$

BY (2).1

(5).2. QED

BY (5).1, $LL1_{\text{NVRAMHistorySummaryUncorruptedEqualsHLAliveLemma}}$

Then, the $LL1_{\text{NVRAMHistorySummaryUncorruptedUnchangedLemma}}$ tells us that the $LL1_{\text{NVRAMHistorySummaryUncorrupted}}$ predicate is unchanged.

(4).3. UNCHANGED $LL1_{\text{NVRAMHistorySummaryUncorrupted}}$

(5).1. $LL1_{\text{TypeInvariant}}$

BY (2).1

The $\text{NVRAM}$ is unchanged by definition of the $LL1_{\text{Restart}}$ action.

(5).2. UNCHANGED $LL1_{\text{NVRAM}}$

BY (3).2

The $UnchangedAuthenticatedHistoryStateBindingsLemma$ tells us that there is no change to the set of history state bindings that have authenticators in the set $LL1_{\text{ObservedAuthenticators}}$.

(5).3. $\forall \text{historyStateBinding} \in \text{HashType} :$

UNCHANGED  $LL1_{\text{HistoryStateBindingAuthenticated}}(\text{historyStateBinding})$

(6).1. UNCHANGED $(LL1_{\text{NVRAM}}, LL1_{\text{ObservedAuthenticators}})$

BY (3).2

(6).2. QED

BY (6).1, $UnchangedAuthenticatedHistoryStateBindingsLemma$

We have all of the antecedents for the $LL1_{\text{NVRAMHistorySummaryUncorruptedUnchangedLemma}}$, so we can apply it directly.

(5).4. QED

BY (5).1, (5).2, (5).3, $LL1_{\text{NVRAMHistorySummaryUncorruptedUnchangedLemma}}$

(4).4. QED

BY (4).1, (4).2, (4).3

The mapping from $LL1_{\text{AvailableInputs}}$ to $HL_{\text{AvailableInputs}}$ is direct.

(3).4. UNCHANGED $HL_{\text{AvailableInputs}}$

(4).1. UNCHANGED $LL1_{\text{AvailableInputs}}$
The mapping from $LL_{1}ObservedOutputs$ to $HLObservedOutputs$ is direct.

We prove the stuttering of the high-level public and private state by using the $NonAdvancementLemma$.

Many of the antecedents for the $NonAdvancementLemma$ come directly from antecedents in the induction.

The $LL_{1}NVRAM$ is unchanged by definition of the $LL_{1}\text{Restart}$ action.

We have all of the antecedents for the $NonAdvancementLemma$, so we can apply it directly.

A Memoir-Basic $LL_{1}\text{ReadDisk}$ action refines to a high-level stuttering step.

The $HLAlive$ predicate is unchanged because the $LL_{1}\text{NVRAMHistorySummaryUncorrupted}$ predicate is unchanged.

The $LL_{1}\text{NVRAMHistorySummaryUncorruptedEqualsHLAliveLemma}$ tells us that $LL_{1}\text{NVRAMHistorySummaryUncorrupted}$ and $HLAlive$ are equal.
(5)2. QED
  BY (5)1, LL1NVRAMHistorySummaryUncorruptedEqualsHLAliveLemma
(4)2. LL1NVRAMHistorySummaryUncorrupted' = HLAlive'
(5)1. LL1Refinement'
  BY (2)1
(5)2. QED
  BY (5)1, LL1NVRAMHistorySummaryUncorruptedEqualsHLAliveLemma

Then, the LL1NVRAMHistorySummaryUncorruptedUnchangedLemma tells us that the LL1NVRAMHistorySummaryUncorrupted predicate is unchanged.

(4)3. UNCHANGED LL1NVRAMHistorySummaryUncorrupted
  (5)1. LL1TypeInvariant
  BY (2)1

The NVRAM is unchanged by definition of the LL1ReadDisk action.

(5)2. UNCHANGED LL1NVRAM
  BY (3)1 DEF LL1ReadDisk

The UnchangedAuthenticatedHistoryStateBindingsLemma tells us that there is no change to the set of history state bindings that have authenticators in the set LL1ObservedAuthenticators.

(5)3. ∀ historyStateBinding ∈ HashType :
    UNCHANGED LL1HistoryStateBindingAuthenticated(historyStateBinding)
  (6)1. UNCHANGED (LL1NVRAM, LL1ObservedAuthenticators)
    BY (3)1 DEF LL1ReadDisk
  (6)2. QED
    BY (6)1, UnchangedAuthenticatedHistoryStateBindingsLemma

We have all of the antecedents for the LL1NVRAMHistorySummaryUncorruptedUnchangedLemma, so we can apply it directly.

(5)4. QED
  BY (5)1, (5)2, (5)3, LL1NVRAMHistorySummaryUncorruptedUnchangedLemma
(4)4. QED
  BY (4)1, (4)2, (4)3

The mapping from LL1AvailableInputs to HLAvalibleInputs is direct.

(3)3. UNCHANGED HLAvalibleInputs
  (4)1. UNCHANGED LL1AvailableInputs
    BY (3)1 DEF LL1ReadDisk
  (4)2. ∧ HLAvalibleInputs = LL1AvailableInputs
    ∧ HLAvalibleInputs' = LL1AvailableInputs'
    BY (2)1 DEF LL1Refinement
  (4)3. QED
    BY (4)1, (4)2

The mapping from LL1ObservedOutputs to HLObservedOutputs is direct.

(3)4. UNCHANGED HLObservedOutputs
  (4)1. UNCHANGED LL1ObservedOutputs
    BY (3)1 DEF LL1ReadDisk
  (4)2. ∧ HLObservedOutputs = LL1ObservedOutputs
    ∧ HLObservedOutputs' = LL1ObservedOutputs'
    BY (2)1 DEF LL1Refinement
  (4)3. QED
    BY (4)1, (4)2

We prove the stuttering of the high-level public and private state by using the NonAdvancementLemma.

(3)5. UNCHANGED (HLPublicState, HLPriavteState)

Many of the antecedents for the NonAdvancementLemma come directly from antecedents in the induction.
1. $LL_1$Refinement
   $LL_1$Refinement'
   $LL_1$TypeInvariant
   $LL_1$TypeInvariant'
   UniquenessInvariant

BY (2)1 DEF CorrectnessInvariants

The NVRAM is unchanged by definition of the $LL_1$ReadDisk action.

2. UNCHANGED $LL_1$NVRAM
   BY (3)1 DEF $LL_1$ReadDisk

The $UnchangedAuthenticatedHistoryStateBindingsLemma$ tells us that there is no change to the set of history state bindings that have authenticators in the set $LL_1$ObservedAuthenticators.

3. $\forall$ historyStateBinding $\in$ HashType :
   UNCHANGED $LL_1$HistoryStateBindingAuthenticated(historyStateBinding)
   (5)1. UNCHANGED ($LL_1$NVRAM, $LL_1$ObservedAuthenticators)
   BY (3)1 DEF $LL_1$ReadDisk
   (5)2. QED
   BY (5)1, $UnchangedAuthenticatedHistoryStateBindingsLemma$

We have all of the antecedents for the NonAdvancementLemma, so we can apply it directly.

4. QED
   BY (4)1, (4)2, (4)3, NonAdvancementLemma

A Memoir-Basic $LL_1$WriteDisk action refines to a high-level stuttering step.

5. $LL_1$WriteDisk $\Rightarrow$ UNCHANGED HLVars
   (3)1. HAVE $LL_1$WriteDisk
   The HLAlive predicate is unchanged because the $LL_1$NVRAMHistorySummaryUncorrupted predicate is unchanged.

6. QED
   BY (3)2, (3)3, (3)4, (3)5 DEF HLVars

A Memoir-Basic $LL_1$WriteDisk action refines to a high-level stuttering step.

A Memoir-Basic $LL_1$WriteDisk action refines to a high-level stuttering step.
UNCHANGED $\text{LL1HistoryStateBindingAuthenticated(historyStateBinding)}$

(6) 1. UNCHANGED $(\text{LL1NVRAM, LL1ObservedAuthenticators})$
    BY (3) 1 DEF $\text{LL1WriteDisk}$

(6) 2. QED
    BY (6) 1, $\text{UnchangedAuthenticatedHistoryStateBindingsLemma}$

We have all of the antecedents for the $\text{LL1NVRAMHistorySummaryUncorruptedUnchangedLemma}$, so we can apply it directly.

(5) 4. QED
    BY (5) 1, (5) 2, (5) 3, $\text{LL1NVRAMHistorySummaryUncorruptedUnchangedLemma}$

The mapping from $\text{LLAvailableInputs}$ to $\text{HLAvailableInputs}$ is direct.

(3) 3. UNCHANGED $\text{HLAvailableInputs}$

(4) 1. UNCHANGED $\text{LLAvailableInputs}$
    BY (3) 1 DEF $\text{LL1WriteDisk}$

(4) 2. $\text{HLAvailableInputs} = \text{LLAvailableInputs}$
    $\text{HLAvailableInputs}' = \text{LLAvailableInputs}'$
    BY (2) 1 DEF $\text{LL1Refinement}$

(4) 3. QED
    BY (4) 1, (4) 2

The mapping from $\text{LLObservedOutputs}$ to $\text{HLObservedOutputs}$ is direct.

(3) 4. UNCHANGED $\text{HLObservedOutputs}$

(4) 1. UNCHANGED $\text{LLObservedOutputs}$
    BY (3) 1 DEF $\text{LL1WriteDisk}$

(4) 2. $\text{HLObservedOutputs} = \text{LLObservedOutputs}$
    $\text{HLObservedOutputs}' = \text{LLObservedOutputs}'$
    BY (2) 1 DEF $\text{LL1Refinement}$

(4) 3. QED
    BY (4) 1, (4) 2

We prove the stuttering of the high-level public and private state by using the $\text{NonAdvancementLemma}$.

(3) 5. UNCHANGED $(\text{HLPublicState, HLPrivateState})$

Many of the antecedents for the $\text{NonAdvancementLemma}$ come directly from antecedents in the induction.

(4) 1. $\land \text{LL1Refinement}$
    $\land \text{LL1Refinement}'$
    $\land \text{LL1TypeInvariant}$
    $\land \text{LL1TypeInvariant'}$
    $\land \text{UniquenessInvariant}$
    BY (2) 1 DEF $\text{CorrectnessInvariants}$

The NVRAM is unchanged by definition of the $\text{LL1WriteDisk}$ action.

(4) 2. UNCHANGED $\text{LL1NVRAM}$
    BY (3) 1 DEF $\text{LL1WriteDisk}$

The $\text{UnchangedAuthenticatedHistoryStateBindingsLemma}$ tells us that there is no change to the set of history state bindings that have authenticators in the set $\text{LL1ObservedAuthenticators}$.

(4) 3. $\forall \text{historyStateBinding} \in \text{HashType} :$
    UNCHANGED $\text{LL1HistoryStateBindingAuthenticated(historyStateBinding)}$

(5) 1. UNCHANGED $(\text{LL1NVRAM, LL1ObservedAuthenticators})$
    BY (3) 1 DEF $\text{LL1WriteDisk}$

(5) 2. QED
    BY (5) 1, $\text{UnchangedAuthenticatedHistoryStateBindingsLemma}$

We have all of the antecedents for the $\text{NonAdvancementLemma}$, so we can apply it directly.
A Memoir-Basic $LL1\text{CorruptRAM}$ action refines to a high-level stuttering step.

(2.9) $LL1\text{CorruptRAM} \Rightarrow \text{UNCHANGED HLVars}$

(3.1) HAVE $LL1\text{CorruptRAM}$

(3.2) PICK $\text{untrustedStorage} \in LL1\text{UntrustedStorageType}$, $\text{fakeSymmetricKey} \in \text{SymmetricKeyType} \setminus \{LL1\text{NVRAM}.\text{symmetricKey}\}$, $\text{hash} \in \text{HashType}$:

$LL1\text{CorruptRAM}(\text{untrustedStorage}, \text{fakeSymmetricKey}, \text{hash})$

BY (3.2) 1 DEF $LL1\text{CorruptRAM}$

The $HL\text{Alive}$ predicate is unchanged because the $LL1\text{NVRAMHistorySummaryUncorrupted}$ predicate is unchanged.

(3.3) UNCHANGED $HL\text{Alive}$

The $LL1\text{NVRAMHistorySummaryUncorruptedEqualsHLAliveLemma}$ tells us that $LL1\text{NVRAMHistorySummaryUncorrupted}$ and $HL\text{Alive}$ are equal.

(4.1) $LL1\text{NVRAMHistorySummaryUncorrupted} = HL\text{Alive}$

(5.1) $LL1\text{Refinement}$

BY (2.1)

(5.2) $QED$

BY (5.1), $LL1\text{NVRAMHistorySummaryUncorruptedEqualsHLAliveLemma}$

(4.2) $LL1\text{NVRAMHistorySummaryUncorrupted}' = HL\text{Alive}'$

(5.1) $LL1\text{Refinement}'$

BY (2.1)

(5.2) $QED$

BY (5.1), $LL1\text{NVRAMHistorySummaryUncorruptedEqualsHLAliveLemma}$

Then, the $LL1\text{NVRAMHistorySummaryUncorruptedUnchangedLemma}$ tells us that the $LL1\text{NVRAMHistorySummaryUncorrupted}$ predicate is unchanged.

(4.3) UNCHANGED $LL1\text{NVRAMHistorySummaryUncorrupted}$

(5.1) $LL1\text{TypeInvariant}$

BY (2.1)

The $LL1\text{NVRAM}$ is unchanged by definition of the $LL1\text{CorruptRAM}$ action.

(5.2) UNCHANGED $LL1\text{NVRAM}$

BY (3.2)

The $UnchangedAuthenticatedHistoryStateBindingsLemma$ tells us that there is no change to the set of history state bindings that have authenticators in the set $LL1\text{ObservedAuthenticators}$.

(5.3) $\forall \text{historyStateBinding} \in \text{HashType}$:

UNCHANGED $LL1\text{HistoryStateBindingAuthenticated}(\text{historyStateBinding})$

(6.1) UNCHANGED $\langle LL1\text{NVRAM}, LL1\text{ObservedAuthenticators} \rangle$

BY (3.2)

(6.2) $QED$

BY (6.1), $UnchangedAuthenticatedHistoryStateBindingsLemma$

We have all of the antecedents for the $LL1\text{NVRAMHistorySummaryUncorruptedUnchangedLemma}$, so we can apply it directly.

(5.4) $QED$

BY (5.1), (5.2), (5.3), $LL1\text{NVRAMHistorySummaryUncorruptedUnchangedLemma}$

(4.4) $QED$

BY (4.1), (4.2), (4.3)

The mapping from $LL1\text{AvailableInputs}$ to $HL\text{AvailableInputs}$ is direct.

(3.4) UNCHANGED $HL\text{AvailableInputs}$

(4.1) UNCHANGED $LL1\text{AvailableInputs}$
BY (3)2
(4)2. ∧ HLAvaliableInputs = LLAvailableInputs
∧ HLAvaliableInputs' = LLAvailableInputs'
BY (2)1 DEF LLRefinement
(4)3. QED
BY (4)1, (4)2
The mapping from LLObservedOutputs to HLObservedOutputs is direct.

(3)5. UNCHANGED HLObservedOutputs
(4)1. UNCHANGED LLObservedOutputs
BY (3)2
(4)2. ∧ HLObservedOutputs = LLObservedOutputs
∧ HLObservedOutputs' = LLObservedOutputs'
BY (2)1 DEF LLRefinement
(4)3. QED
BY (4)1, (4)2
We prove the stuttering of the high-level public and private state by using the NonAdvancementLemma.

(3)6. UNCHANGED ⟨HLPublicState, HLPriereatState⟩
Many of the antecedents for the NonAdvancementLemma come directly from antecedents in the induction.

(4)1. ∧ LLRefinement
∧ LLRefinement'
∧ LLTypeInvariant
∧ LLTypeInvariant'
∧ UniquenessInvariant
BY (2)1 DEF CorrectnessInvariants
The NVRAM is unchanged by definition of the LL1CorruptRAM action.

(4)2. UNCHANGED LL1NVRAM
BY (3)2
The UnchangedAuthenticatedHistoryStateBindingsLemma tells us that there is no change to the set of history state bindings that have authenticators in the set LL1ObservedAuthenticators.

(4)3. ∀ historyStateBinding ∈ HashType :
UNCHANGED LL1HistoryStateBindingAuthenticated(historyStateBinding)
(5)1. UNCHANGED ⟨LL1NVRAM, LL1ObservedAuthenticators⟩
BY (3)2
(5)2. QED
BY (5)1, UnchangedAuthenticatedHistoryStateBindingsLemma
We have all of the antecedents for the NonAdvancementLemma, so we can apply it directly.

(4)4. QED
BY (4)1, (4)2, (4)3, NonAdvancementLemma
(3)7. QED
BY (3)3, (3)4, (3)5, (3)6 DEF HLVars

A Memoir-Basic LL1RestrictedCorruption action refines to a high-level HLDie step.

(2)10. LL1RestrictedCorruption ⇒ HLDie
(3)1. HAVE LL1RestrictedCorruption
(3)2. PICK garbageHistorySummary ∈ HashType :
LL1RestrictedCorruption!nvram!garbageHistorySummary
BY (3)1 DEF LL1RestrictedCorruption
First, we prove that this action causes the LL1NVRAMHistorySummaryUncorrupted predicate to become false.

(3)3. LL1NVRAMHistorySummaryUncorrupted' = FALSE
We will make use of the conjunct in LL1RestrictedCorruption that prevents the garbage history summary from being in an authenticated history state binding. This conjunct states a 2-way universally quantified predicate.
4.1. \( LL1_{\text{RestrictedCorruption}!nvr! current(garbageHistorySummary)} \)
   \( \text{by (3)2} \)

The following equivalence, plus the knowledge that the symmetric key in the NVRAM and the set of observed authenticators have not changed, are sufficient to prove that the above 2-way universally quantified predicate equals the negation of the 2-way universally quantified predicate in \( LL1_{\text{NVRAMHistorySummaryUncorrupted}} \), when expanded through \( LL1_{\text{HistoryStateBindingAuthenticated}} \).

4.2. \( LL1_{\text{NVRAM.historySummary}'} = garbageHistorySummary \)

5.1. \( LL1_{\text{NVRAM}'} = [\text{historySummary} \mapsto garbageHistorySummary, \symmetricKey \mapsto LL1_{\text{NVRAM}.symmetricKey}] \)
   \( \text{by (3)2} \)

5.2. QED
   \( \text{by (5)1} \)

4.3. UNCHANGED \( LL1_{\text{NVRAM}.symmetricKey} \)
   \( \text{by (2)1, SymmetricKeyConstantLemma} \)

4.4. UNCHANGED \( LL1_{\text{ObservedAuthenticators}} \)
   \( \text{by (3)1 DEF LL1_{\text{RestrictedCorruption}} } \)

5. QED
   \( \text{by (4)1, (4)2, (4)3, (4)4 DEF LL1_{\text{NVRAMHistorySummaryUncorrupted}, LL1_{\text{HistoryStateBindingAuthenticated}}} } \)

Because the \( LL1_{\text{NVRAMHistorySummaryUncorrupted}} \) is false, the \( LL1_{\text{Refinement}} \) immediately tells us that the high-level system is not alive.

3.4. \( HLA_{\text{live}'} = \text{false} \)
   \( \text{by (2)1, (3)3 DEF LL1_{\text{Refinement}} } \)

The mapping from \( LL1_{\text{AvailableInputs}} \) to \( HLA_{\text{AvailableInputs}} \) is direct.

3.5. UNCHANGED \( HLA_{\text{AvailableInputs}} \)
   \( \text{by (4)1, UNCHANGED LL1_{\text{AvailableInputs}} } \)
   \( \text{by (3)1 DEF LL1_{\text{RestrictedCorruption}} } \)

4.2. \( \land HLA_{\text{AvailableInputs}} = LL1_{\text{AvailableInputs}} \)
   \( \land HLA_{\text{AvailableInputs}'} = LL1_{\text{AvailableInputs}'} \)
   \( \text{by (2)1 DEF LL1_{\text{Refinement}} } \)

4.3. QED
   \( \text{by (4)1, (4)2 } \)

The mapping from \( LL1_{\text{ObservedOutputs}} \) to \( HLO_{\text{ObservedOutputs}} \) is direct.

3.6. UNCHANGED \( HLO_{\text{ObservedOutputs}} \)
   \( \text{by (4)1, UNCHANGED LL1_{\text{ObservedOutputs}} } \)
   \( \text{by (3)1 DEF LL1_{\text{RestrictedCorruption}} } \)

4.2. \( \land HLO_{\text{ObservedOutputs}} = LL1_{\text{ObservedOutputs}} \)
   \( \land HLO_{\text{ObservedOutputs}'} = LL1_{\text{ObservedOutputs}'} \)
   \( \text{by (2)1 DEF LL1_{\text{Refinement}} } \)

4.3. QED
   \( \text{by (4)1, (4)2 } \)

Because the \( LL1_{\text{NVRAMHistorySummaryUncorrupted}} \) is false, the \( LL1_{\text{Refinement}} \) immediately tells us that the high-level public state equals the dead state.

3.7. \( HLP_{\text{PublicState}'} = \text{DeadPublicState} \)
   \( \text{by (2)1, (3)3 DEF LL1_{\text{Refinement}} } \)

Because the \( LL1_{\text{NVRAMHistorySummaryUncorrupted}} \) is false, the \( LL1_{\text{Refinement}} \) immediately tells us that the high-level private state equals the dead state.

3.8. \( HLP_{\text{PrivateState}'} = \text{DeadPrivateState} \)
   \( \text{by (2)1, (3)3 DEF LL1_{\text{Refinement}} } \)

3.9. QED
   \( \text{by (3)4, (3)5, (3)6, (3)7, (3)8 DEF HLDie } \)
Using the \textit{StepSimulation} proof rule, the base case and the induction step together imply that the invariant always holds.

\[\Box[LL_{1\text{Next}}]_{LL_{1\text{Vars}}} \land \Box LL_{1\text{Refinement}} \land \Box CorrectnessInvariants \Rightarrow \Box[HL_{\text{Next}}]_{HL_{\text{Vars}}}\]

By (1)4, \textit{StepSimulation}

\[\text{(2)2. QED}
\text{BY (1)1, (1)2, (1)3, (2)1 DEF } LL_{1\text{Spec}}, HLSpec, LL_{1\text{Refinement}}\]
This module states and proves several lemmas that are useful for proving type safety. Since type safety is an important part of the implementation proof, these lemmas also will be used in theorems other than the Memoir-Opt type-safety theorem.

The lemmas in this module are:
- CheckpointDefsTypeSafeLemma
- CheckpointTypeSafe
- SuccessorDefsTypeSafeLemma
- SuccessorTypeSafe
- LL2SubtypeImplicationLemma
- LL2InitDefsTypeSafeLemma
- LL2PerformOperationDefsTypeSafeLemma
- LL2RepeatOperationDefsTypeSafeLemma
- LL2TakeCheckpointDefsTypeSafeLemma
- LL2CorruptSPCRDefsTypeSafeLemma
- AuthenticatorsMatchDefsTypeSafeLemma

EXTENDS MemoirLL2Refinement

The CheckpointDefsTypeSafeLemma proves that the definitions within the \texttt{let} of the Checkpoint function all have the appropriate type. This is a trivial proof that merely walks through the definitions.

THEOREM CheckpointDefsTypeSafeLemma \(\Delta\)

\[\forall \text{historySummary} \in \text{HistorySummaryType} :\]
\[
\text{LET}
\]
\[
\text{checkpointedAnchor} \ \triangleq \ \text{Hash}(\text{historySummary}.\text{anchor}, \text{historySummary}.\text{extension})
\]
\[
\text{checkpointedHistorySummary} \ \triangleq \ \left\langle \begin{array}{c}
\text{anchor} \mapsto \text{checkpointedAnchor}, \\
\text{extension} \mapsto \text{BaseHashValue}
\end{array} \right\rangle
\]
\[
\text{IN}
\]
\[
\wedge \text{checkpointedAnchor} \in \text{HashType}
\]
\[
\wedge \text{checkpointedHistorySummary} \in \text{HistorySummaryType}
\]
\[
\wedge \text{checkpointedHistorySummary}.\text{anchor} \in \text{HashType}
\]
\[
\wedge \text{checkpointedHistorySummary}.\text{extension} \in \text{HashType}
\]

\{1\} 1. TAKE \text{historySummary} \in \text{HistorySummaryType}
\{1\} checkpointedAnchor \(\triangleq\) \text{Hash}(\text{historySummary}.\text{anchor}, \text{historySummary}.\text{extension})
\{1\} checkpointedHistorySummary \(\triangleq\) \left\langle \begin{array}{c}
\text{anchor} \mapsto \text{checkpointedAnchor}, \\
\text{extension} \mapsto \text{BaseHashValue}
\end{array} \right\rangle
\{1\} HIDE DEF checkpointedAnchor, checkpointedHistorySummary
\{1\} checkpointedAnchor \in \text{HashType}

\{2\} 1. \text{historySummary}.\text{anchor} \in \text{HashDomain}
\{3\} 1. \text{historySummary}.\text{anchor} \in \text{HashType}
\quad \text{BY} \{1\} 1 \text{DEF HistorySummaryType}
\{3\} 2. QED
\quad \text{BY} \{3\} 1 \text{DEF HashDomain}
\{2\} 2. \text{historySummary}.\text{extension} \in \text{HashDomain}
\{3\} 1. \text{historySummary}.\text{extension} \in \text{HashType}
\quad \text{BY} \{1\} 1 \text{DEF HistorySummaryType}
\{3\} 2. QED
\quad \text{BY} \{3\} 1 \text{DEF HashDomain}
\{2\} 3. QED
Theorem \( \text{CheckpointTypeSafe} \) proves that the definitions within the \texttt{let} of the \texttt{Successor} function all have the appropriate type. This is a trivial proof that merely walks through the definitions.

Theorem \( \text{SuccessorDefsTypeSafeLemma} \) states that for all \( \texttt{HistorySummary} \in \text{HistorySummaryType} \), \( \texttt{Input} \in \text{InputType} \), and \( \texttt{hashBarrier} \in \text{HashType} \):
\[ \begin{align*}
&\text{\(\wedge\ newHistorySummary \in HistorySummaryType\)} \\
&\text{\(\wedge\ newHistorySummary.\text{anchor} \in HashType\)} \\
&\text{\(\wedge\ newHistorySummary.\text{extension} \in HashType\)} \\
\end{align*} \]

\{1\}1. \text{TAKETake}\ historySummary \in HistorySummaryType, \text{input} \in InputType, \text{hashBarrier} \in HashType

\{1\} \text{securedInput} \triangleq \text{Hash}(\text{hashBarrier}, \text{input})

\{1\} \text{newAnchor} \triangleq \text{historySummary.\text{anchor}}

\{1\} \text{newExtension} \triangleq \text{Hash}(\text{historySummary.\text{extension}}, \text{securedInput})

\{1\} \text{newHistorySummary} \triangleq [\text{anchor} \mapsto \text{newAnchor}, \text{extension} \mapsto \text{newExtension}]

\{1\} \text{HIDE DEF } securedInput, \text{newAnchor}, \text{newExtension}, \text{newHistorySummary}

\{1\}2. \text{securedInput} \in HashType

\{2\}1. \text{hashBarrier} \in HashDomain

\{3\}1. \text{hashBarrier} \in HashType

\text{BY } \{1\}1

\{3\}2. \text{QED}

\text{BY } \{3\}1 \text{ DEF } HashDomain

\{2\}2. \text{input} \in HashDomain

\{3\}1. \text{input} \in InputType

\text{BY } \{1\}1

\{3\}2. \text{QED}

\text{BY } \{3\}1 \text{ DEF } HashDomain

\{2\}3. \text{QED}

\text{BY } \{2\}1, \{2\}2, \text{HashTypeSafe DEF } securedInput

\{1\}3. \text{newAnchor} \in HashType

\{2\}1. \text{historySummary} \in HistorySummaryType

\text{BY } \{1\}1

\{2\}2. \text{QED}

\text{BY } \{2\}1 \text{ DEF } newAnchor, HistorySummaryType

\{1\}4. \text{newExtension} \in HashType

\{2\}1. \text{historySummary.\text{extension}} \in HashDomain

\{3\}1. \text{historySummary.\text{extension}} \in HashType

\{4\}1. \text{historySummary} \in HistorySummaryType

\text{BY } \{1\}1

\{4\}2. \text{QED}

\text{BY } \{4\}1 \text{ DEF } HistorySummaryType

\{3\}2. \text{QED}

\text{BY } \{3\}1 \text{ DEF } HashDomain

\{2\}2. \text{securedInput} \in HashDomain

\text{BY } \{1\}2 \text{ DEF } HashDomain

\{2\}3. \text{QED}

\text{BY } \{2\}1, \{2\}2, \text{HashTypeSafe DEF } newExtension

\{1\}5. \text{newHistorySummary} \in HistorySummaryType

\text{BY } \{1\}3, \{1\}4 \text{ DEF } newHistorySummary, HistorySummaryType

\{1\}6. \wedge \text{newHistorySummary.\text{anchor}} \in HashType

\wedge \text{newHistorySummary.\text{extension}} \in HashType

\text{BY } \{1\}5 \text{ DEF } HistorySummaryType

\{1\}7. \text{QED}

\text{BY } \{1\}2, \{1\}3, \{1\}4, \{1\}5, \{1\}6

\text{DEF } securedInput, newAnchor, newExtension, newHistorySummary

\begin{center}
\text{Type safety of the Successor function.}
\end{center}
THEOREM SuccessorTypeSafe \( \triangleq \)
\[ \forall \text{historySummary} \in \text{HistorySummaryType}, \text{input} \in \text{InputType}, \text{hashBarrier} \in \text{HashType} : \]
\[ \text{Successor}(\text{historySummary}, \text{input}, \text{hashBarrier}) \in \text{HistorySummaryType} \]
\[ \langle 1 \rangle. \text{take} \text{historySummary} \in \text{HistorySummaryType}, \text{input} \in \text{InputType}, \text{hashBarrier} \in \text{HashType} \]
\[ \langle 1 \rangle \text{securedInput} \triangleq \text{Hash}(\text{hashBarrier}, \text{input}) \]
\[ \langle 1 \rangle \text{newAnchor} \triangleq \text{historySummary}.\text{anchor} \]
\[ \langle 1 \rangle \text{newExtension} \triangleq \text{Hash}(\text{historySummary}.\text{extension}, \text{securedInput}) \]
\[ \langle 1 \rangle \text{newHistorySummary} \triangleq [ \]
\[ \text{anchor} \mapsto \text{newAnchor}, \]
\[ \text{extension} \mapsto \text{newExtension} ] \]
\[ \langle 1 \rangle 2. \land \text{securedInput} \in \text{HashType} \]
\[ \land \text{newAnchor} \in \text{HashType} \]
\[ \land \text{newExtension} \in \text{HashType} \]
\[ \land \text{newHistorySummary} \in \text{HistorySummaryType} \]
\[ \land \text{newHistorySummary}.\text{anchor} \in \text{HashType} \]
\[ \land \text{newHistorySummary}.\text{extension} \in \text{HashType} \]
\[ \text{by} \ (1) 1, \text{SuccessorDefsTypeSafeLemma} \]
\[ \langle 1 \rangle 3. \text{Successor}(\text{historySummary}, \text{input}, \text{hashBarrier}) = \text{newHistorySummary} \]
\[ \text{by} \ (1) 1 \text{def} \text{Successor}, \text{newHistorySummary}, \text{newExtension}, \text{newAnchor}, \text{securedInput} \]
\[ \langle 1 \rangle 4. \text{newHistorySummary} \in \text{HistorySummaryType} \]
\[ \text{by} \ (1) 2 \]
\[ \langle 1 \rangle 5. \text{QED} \]
\[ \text{by} \ (1) 3, (1) 4 \]

The \text{LL2SubtypeImplicationLemma} proves that when the \text{LL2TypeInvariant} holds, the subtypes of \text{LL2Disk}, \text{LL2RAM}, and \text{LL2NVRAM} also hold. This is asserted and proven for both the unprimed and primed states.

The proof itself is completely trivial. It follows directly from the type definitions \text{LL2UntrustedStorageType} and \text{LL2TrustedStorageType}.

\text{LL2SubtypeImplication} \( \triangleq \)
\[ \text{LL2TypeInvariant} \Rightarrow \]
\[ \land \text{LL2Disk}.\text{publicState} \in \text{PublicStateType} \]
\[ \land \text{LL2Disk}.\text{privateStateEnc} \in \text{PrivateStateEncType} \]
\[ \land \text{LL2Disk}.\text{historySummary} \in \text{HistorySummaryType} \]
\[ \land \text{LL2Disk}.\text{historySummary}.\text{anchor} \in \text{HashType} \]
\[ \land \text{LL2Disk}.\text{authentication} \in \text{MACType} \]
\[ \land \text{LL2RAM}.\text{publicState} \in \text{PublicStateType} \]
\[ \land \text{LL2RAM}.\text{privateStateEnc} \in \text{PrivateStateEncType} \]
\[ \land \text{LL2RAM}.\text{historySummary} \in \text{HistorySummaryType} \]
\[ \land \text{LL2RAM}.\text{historySummary}.\text{anchor} \in \text{HashType} \]
\[ \land \text{LL2RAM}.\text{historySummary}.\text{extension} \in \text{HashType} \]
\[ \land \text{LL2RAM}.\text{authentication} \in \text{MACType} \]
\[ \land \text{LL2NVRAM}.\text{historySummaryAnchor} \in \text{HashType} \]
\[ \land \text{LL2NVRAM}.\text{symmetricKey} \in \text{SymmetricKeyType} \]
\[ \land \text{LL2NVRAM}.\text{hashBarrier} \in \text{HashType} \]
\[ \land \text{LL2NVRAM}.\text{extensionInProgress} \in \text{BOOLEAN} \]

THEOREM \text{LL2SubtypeImplicationLemma} \( \triangleq \)
\[ \land \text{LL2SubtypeImplication} \]
\[ \land \text{LL2SubtypeImplication}' \]
\[ \langle 1 \rangle 1. \text{LL2SubtypeImplication} \]

195
(2). SUFFICES

ASSUME LL2TypeInvariant

PROVE

\[ \wedge LL2Disk.publicState \in PublicState.Type \]
\[ \wedge LL2Disk.privateStateEnc \in PrivateStateEnc.Type \]
\[ \wedge LL2Disk.historySummary \in HistorySummary.Type \]
\[ \wedge LL2Disk.historySummary.anchor \in Hash.Type \]
\[ \wedge LL2Disk.historySummary.extension \in Hash.Type \]
\[ \wedge LL2Disk.authenticator \in MAC.Type \]
\[ \wedge LL2RAM.publicState \in PublicState.Type \]
\[ \wedge LL2RAM.privateStateEnc \in PrivateStateEnc.Type \]
\[ \wedge LL2RAM.historySummary \in HistorySummary.Type \]
\[ \wedge LL2RAM.historySummary.anchor \in Hash.Type \]
\[ \wedge LL2RAM.historySummary.extension \in Hash.Type \]
\[ \wedge LL2RAM.authenticator \in MAC.Type \]
\[ \wedge LL2NVRAM.historySummaryAnchor \in Hash.Type \]
\[ \wedge LL2NVRAM.symmetricKey \in SymmetricKey.Type \]
\[ \wedge LL2NVRAM.hashBarrier \in Hash.Type \]
\[ \wedge LL2NVRAM.extensionInProgress \in BOOLEAN \]

BY DEF LL2SubtypeImplication, LL2TypeInvariant

(2).2. LL2Disk \in LL2UntrustedStorage.Type

BY (2).1 DEF LL2TypeInvariant

(2).3. LL2RAM \in LL2UntrustedStorage.Type

BY (2).1 DEF LL2TypeInvariant

(2).4. LL2NVRAM \in LL2TrustedStorage.Type

BY (2).1 DEF LL2TypeInvariant

(2).5. LL2Disk.publicState \in PublicState.Type

BY (2).2 DEF LL2UntrustedStorage.Type

(2).6. LL2Disk.privateStateEnc \in PrivateStateEnc.Type

BY (2).2 DEF LL2UntrustedStorage.Type

(2).7. LL2Disk.historySummary \in HistorySummary.Type

BY (2).2 DEF LL2UntrustedStorage.Type

(2).8. LL2Disk.historySummary.anchor \in Hash.Type

BY (2).7 DEF HistorySummary.Type

(2).9. LL2Disk.historySummary.extension \in Hash.Type

BY (2).7 DEF HistorySummary.Type

(2).10. LL2Disk.authenticator \in MAC.Type

BY (2).2 DEF LL2UntrustedStorage.Type

(2).11. LL2RAM.publicState \in PublicState.Type

BY (2).3 DEF LL2UntrustedStorage.Type

(2).12. LL2RAM.privateStateEnc \in PrivateStateEnc.Type

BY (2).3 DEF LL2UntrustedStorage.Type

(2).13. LL2RAM.historySummary \in HistorySummary.Type

BY (2).3 DEF LL2UntrustedStorage.Type

(2).14. LL2RAM.historySummary.anchor \in Hash.Type

BY (2).13 DEF HistorySummary.Type

(2).15. LL2RAM.historySummary.extension \in Hash.Type

BY (2).13 DEF HistorySummary.Type

(2).16. LL2RAM.authenticator \in MAC.Type

BY (2).3 DEF LL2UntrustedStorage.Type

(2).17. LL2NVRAM.historySummaryAnchor \in Hash.Type

BY (2).4 DEF LL2TrustedStorage.Type
(2)18. $LL2NVRAM.symmetricKey \in SymmetricKeyType$
   BY (2)4 DEF $LL2TrustedStorageType$
(2)19. $LL2NVRAM.hashBarrier \in HashType$
   BY (2)4 DEF $LL2TrustedStorageType$
(2)20. $LL2NVRAM.extensionInProgress \in BOOLEAN$
   BY (2)4 DEF $LL2TrustedStorageType$
(2)21. QED
   BY (2)5, (2)6, (2)7, (2)8, (2)9, (2)10, (2)11, (2)12,
       (2)13, (2)14, (2)15, (2)16, (2)17, (2)18, (2)19, (2)20

(1)2. $LL2SubtypeImplication'$
(2)1. SUFFICES
   ASSUME $LL2TypeInvariant'$

   PROVE
   $\forall LL2Disk.publicState' \in PublicStateType$
   $\forall LL2Disk.privateStateEnc' \in PrivateStateEncType$
   $\forall LL2Disk.historySummary' \in HistorySummaryType$
   $\forall LL2Disk.historySummary.anchor' \in HashType$
   $\forall LL2Disk.historySummary.extension' \in HashType$
   $\forall LL2Disk.authenticator' \in MACType$
   $\forall LL2RAM.publicState' \in PublicStateType$
   $\forall LL2RAM.privateStateEnc' \in PrivateStateEncType$
   $\forall LL2RAM.historySummary' \in HistorySummaryType$
   $\forall LL2RAM.historySummary.anchor' \in HashType$
   $\forall LL2RAM.historySummary.extension' \in HashType$
   $\forall LL2RAM.authenticator' \in MACType$
   $\forall LL2NVRAM.historySummary.Anchor' \in HashType$
   $\forall LL2NVRAM.symmetricKey' \in SymmetricKeyType$
   $\forall LL2NVRAM.hashBarrier' \in HashType$
   $\forall LL2NVRAM.extensionInProgress' \in BOOLEAN$

BY DEF $LL2SubtypeImplication', LL2TypeInvariant$
(2)2. $LL2Disk' \in LL2UntrustedStorageType$
   BY (2)1 DEF $LL2TypeInvariant$
(2)3. $LL2RAM' \in LL2UntrustedStorageType$
   BY (2)1 DEF $LL2TypeInvariant$
(2)4. $LL2NVRAM' \in LL2TrustedStorageType$
   BY (2)1 DEF $LL2TypeInvariant$
(2)5. $LL2Disk.publicState' \in PublicStateType$
   BY (2)2 DEF $LL2UntrustedStorageType$
(2)6. $LL2Disk.privateStateEnc' \in PrivateStateEncType$
   BY (2)2 DEF $LL2UntrustedStorageType$
(2)7. $LL2Disk.historySummary' \in HistorySummaryType$
   BY (2)2 DEF $LL2UntrustedStorageType$
(2)8. $LL2Disk.historySummary.anchor' \in HashType$
   BY (2)7 DEF $HistorySummaryType$
(2)9. $LL2Disk.historySummary.extension' \in HashType$
   BY (2)7 DEF $HistorySummaryType$
(2)10. $LL2Disk.authenticator' \in MACType$
   BY (2)2 DEF $LL2UntrustedStorageType$
(2)11. $LL2RAM.publicState' \in PublicStateType$
   BY (2)3 DEF $LL2UntrustedStorageType$
(2)12. $LL2RAM.privateStateEnc' \in PrivateStateEncType$
   BY (2)3 DEF $LL2UntrustedStorageType$
The LL2RAM.historySummary' ∈ HistorySummaryType
by (2)(3) DEF LL2UntrustedStorageType
(2)(4) LL2RAM.historySummary, anchor' ∈ HashType
by (2)(5) DEF HistorySummaryType
(2)(6) LL2RAM.historySummary, extension' ∈ HashType
by (2)(5) DEF HistorySummaryType
(2)(7) LL2RAM.authenticator' ∈ MACType
by (2)(5) DEF LL2UntrustedStorageType
(2)(8) LL2NVRAM.historySummaryAnchor' ∈ HashType
by (2)(8) DEF LL2TrustedStorageType
(2)(9) LL2NVRAM.symmetricKey' ∈ SymmetricKeyType
by (2)(8) DEF LL2TrustedStorageType
(2)(10) LL2NVRAM.hashBarrier' ∈ HashType
by (2)(8) DEF LL2TrustedStorageType
(2)(11) LL2NVRAM.extensionInProgress' ∈ BOOLEAN
by (2)(8) DEF LL2TrustedStorageType
(2)(12) QED
by (2)(1), (2)(2)

The LL2InitDefsTypeSafeLemma proves that the definitions within the LET of the LL2Init action all have the appropriate type. This is a trivial proof that merely walks through the definitions.

THEOREM LL2InitDefsTypeSafeLemma ≜
∀ symmetricKey ∈ SymmetricKeyType, hashBarrier ∈ HashType :
LET
  initialPrivateStateEnc ≜ SymmetricEncrypt(symmetricKey, InitialPrivateState)
  initialStateHash ≜ Hash(InitialPublicState, initialPrivateStateEnc)
  initialHistorySummary ≜ [ anchor ⇒ BaseHashValue, extension ⇒ BaseHashValue ]
  initialHistorySummaryHash ≜ Hash(BaseHashValue, BaseHashValue)
  initialHistoryStateBinding ≜ Hash(initialHistorySummaryHash, initialStateHash)
  initialAuthenticator ≜ GenerateMAC(symmetricKey, initialHistoryStateBinding)
  initialUntrustedStorage ≜ [ publicState ⇒ InitialPublicState, privateStateEnc ⇒ initialPrivateStateEnc, historySummary ⇒ initialHistorySummary, authenticator ⇒ initialAuthenticator ]
  initialTrustedStorage ≜ [ historySummaryAnchor ⇒ BaseHashValue, symmetricKey ⇒ symmetricKey, hashBarrier ⇒ hashBarrier, extensionInProgress ⇒ FALSE ]
IN
  ∧ initialPrivateStateEnc ∈ PrivateStateEncType
  ∧ initialStateHash ∈ HashType
  ∧ initialHistorySummary ∈ HistorySummaryType
  ∧ initialHistorySummaryHash ∈ HashType
  ∧ initialHistoryStateBinding ∈ HashType
  ∧ initialAuthenticator ∈ MACType

198
\[ \text{initialUntrustedStorage} \in \text{LL2UntrustedStorageType} \land \text{initialTrustedStorage} \in \text{LL2TrustedStorageType} \]

\[ \begin{array}{l}
\langle 1 \rangle \text{. } \text{TAKESymmetricKey} \in \text{SymmetricKeyType}, \text{hashBarrier} \in \text{HashType} \\
\langle 1 \rangle \text{. } \text{initialPrivateStateEnc} \triangleq \text{SymmetricEncrypt(symmetricKey, InitialPrivateState)} \\
\langle 1 \rangle \text{. } \text{initialStateHash} \triangleq \text{Hash(InitialPublicState, initialPrivateStateEnc)} \\
\langle 1 \rangle \text{. } \text{initialHistorySummary} \triangleq \{ \\
\quad \text{anchor} \mapsto \text{BaseHashValue,} \\
\quad \text{extension} \mapsto \text{BaseHashValue} \} \\
\langle 1 \rangle \text{. } \text{initialHistorySummaryHash} \triangleq \text{Hash(BaseHashValue, BaseHashValue) } \\
\langle 1 \rangle \text{. } \text{initialHistoryStateBinding} \triangleq \text{Hash(initialHistorySummaryHash, initialStateHash)} \\
\langle 1 \rangle \text{. } \text{initialAuthenticator} \triangleq \text{GenerateMAC(symmetricKey, initialHistoryStateBinding)} \\
\langle 1 \rangle \text{. } \text{initialUntrustedStorage} \triangleq \{ \\
\quad \text{publicState} \mapsto \text{InitialPublicState,} \\
\quad \text{privateStateEnc} \mapsto \text{initialPrivateStateEnc,} \\
\quad \text{historySummary} \mapsto \text{initialHistorySummary,} \\
\quad \text{authenticator} \mapsto \text{initialAuthenticator} \} \\
\langle 1 \rangle \text{. } \text{initialTrustedStorage} \triangleq \{ \\
\quad \text{historySummaryAnchor} \mapsto \text{BaseHashValue,} \\
\quad \text{symmetricKey} \mapsto \text{symmetricKey,} \\
\quad \text{hashBarrier} \mapsto \text{hashBarrier,} \\
\quad \text{extensionInProgress} \mapsto \text{FALSE} \} \\
\end{array} \]

\[ \text{HIDE DEF initialPrivateStateEnc, initialStateHash, initialHistorySummary}, \]
\[ \text{initialHistorySummaryHash, initialHistoryStateBinding, initialAuthenticator,} \]
\[ \text{initialUntrustedStorage, initialTrustedStorage} \]

\[ \begin{array}{l}
\langle 1 \rangle \text{. } \text{initialPrivateStateEnc} \in \text{PrivateStateEncType} \\
\langle 2 \rangle \text{. } \text{symmetricKey} \in \text{SymmetricKeyType} \\
\text{BY } \langle 1 \rangle \text{. } \text{1} \\
\langle 2 \rangle \text{. } \text{InitialPrivateState} \in \text{PrivateStateType} \\
\text{BY } \text{ConstantsTypeSafe} \\
\langle 2 \rangle \text{. } \text{QED} \\
\text{BY } \langle 2 \rangle \text{. } \langle 2 \rangle \text{. } \text{SymmetricEncryptionTypeSafe} \text{DEF initialPrivateStateEnc} \\
\langle 1 \rangle \text{. } \text{initialStateHash} \in \text{HashType} \\
\langle 2 \rangle \text{. } \text{InitialPublicState} \in \text{HashDomain} \\
\langle 3 \rangle \text{. } \text{InitialPublicState} \in \text{PublicStateType} \\
\text{BY } \text{ConstantsTypeSafe} \\
\langle 3 \rangle \text{. } \text{QED} \\
\text{BY } \langle 3 \rangle \text{. } \langle 1 \rangle \text{. } \text{1} \text{ DEF HashDomain} \\
\langle 2 \rangle \text{. } \text{initialPrivateStateEnc} \in \text{HashDomain} \\
\text{BY } \langle 1 \rangle \text{. } \langle 2 \rangle \text{. } \text{DEF HashDomain} \\
\langle 2 \rangle \text{. } \text{QED} \\
\text{BY } \langle 2 \rangle \text{. } \langle 2 \rangle \text{. } \text{HashTypeSafe} \text{DEF initialStateHash} \\
\langle 1 \rangle \text{. } \text{initialHistorySummary} \in \text{HistorySummaryType} \\
\langle 2 \rangle \text{. } \text{BaseHashValue} \in \text{HashType} \\
\text{BY } \text{BaseHashValueTypeSafe} \\
\langle 2 \rangle \text{. } \text{QED} \\
\text{BY } \langle 2 \rangle \text{. } \langle 1 \rangle \text{. } \text{1} \text{ DEF HistorySummaryType} \\
\langle 1 \rangle \text{. } \text{initialHistorySummaryHash} \in \text{HashType} \\
\langle 2 \rangle \text{. } \text{BaseHashValue} \in \text{HashDomain} \\
\langle 3 \rangle \text{. } \text{BaseHashValue} \in \text{HashType} \\
\text{BY } \text{BaseHashValueTypeSafe} \\
\langle 3 \rangle \text{. } \text{QED} \\
\text{BY } \langle 3 \rangle \text{. } \langle 1 \rangle \text{. } \text{1} \text{ DEF HashDomain} \]
\( \text{QED} \) by (2)1, HashTypeSafe
\( \text{initialHistorySummaryHash} \)

(2)2. QED

by (2)1, HashTypeSafe
\( \text{def initialHistoryStateBinding} \)

(1)6. initialHistoryStateBinding \in HashType

(2)1. initialHistorySummaryHash \in HashDomain

by (1)5 def HashDomain

(2)2. initialStateHash \in HashDomain

by (1)3 def HashDomain

(2)3. QED

by (2)1, (2)2, HashTypeSafe
\( \text{def initialHistoryStateBinding} \)

(1)7. initialAuthenticator \in MACType

(2)1. symmetricKey \in SymmetricKeyType

by (1)1

(2)2. initialHistoryStateBinding \in HashType

by (1)6

(2)3. QED

by (2)1, (2)2, GenerateMACTypeSafe
\( \text{def initialAuthenticator} \)

(1)8. initialUntrustedStorage \in LL2UntrustedStorageType

(2)1. InitialPublicState \in PublicStateType

by ConstantsTypeSafe

(2)2. initialPrivateKeyEnc \in PrivateStateEncType

by (1)2

(2)3. initialHistorySummary \in HistorySummaryType

by (1)4

(2)4. initialAuthenticator \in MACType

by (1)7

(2)5. QED

by (2)1, (2)2, (2)3, (2)4 def initialUntrustedStorage, LL2UntrustedStorageType

(1)9. initialTrustedStorage \in LL2TrustedStorageType

(2)1. BaseHashValue \in HashType

by BaseHashValueTypeSafe

(2)2. symmetricKey \in SymmetricKeyType

by (1)1

(2)3. hashBarrier \in HashType

by (1)1

(2)4. FALSE \in BOOLEAN

obvious

(2)5. QED

by (2)1, (2)2, (2)3, (2)4 def initialTrustedStorage, LL2TrustedStorageType

(1)10. QED

by (1)2, (1)3, (1)4, (1)5, (1)6, (1)7, (1)8, (1)9

def initialPrivateKeyEnc, initialStateHash, initialHistorySummary, initialHistorySummaryHash, initialHistoryStateBinding, initialAuthenticator, initialUntrustedStorage, initialTrustedStorage

The \( \text{LL2PerformOperationDefsTypeSafeLemma} \) proves that the definitions within the \texttt{let} of the \( \text{LL2PerformOperation} \) action all have the appropriate type. This is a trivial proof that merely walks through the definitions.

\textbf{Theorem} \( \text{LL2PerformOperationDefsTypeSafeLemma} \) \( \triangleq \)

\(\forall \text{input} \in \text{LL2AvailableInputs :} \)

\( \text{LL2TypeInvariant} \Rightarrow \)

\( \text{let} \)

\( \text{historySummaryHash} \triangleq \)

\( \text{Hash} (\text{LL2RAM.historySummary.anchor, LL2RAM.historySummary.extension}) \)
\[
\begin{align*}
\text{stateHash} & \triangleq \text{Hash}(LL2\text{RAM}.\text{publicState}, LL2\text{RAM}.\text{privateStateEnc}) \\
\text{historyStateBinding} & \triangleq \text{Hash(\text{historySummaryHash}, \text{stateHash})} \\
\text{privateState} & \triangleq \text{SymmetricDecrypt(LL2\text{NVRAM}.\text{symmetricKey}, LL2\text{RAM}.\text{privateStateEnc})} \\
\text{sResult} & \triangleq \text{Service(LL2\text{RAM}.\text{publicState}, \text{privateState}, \text{input})} \\
\text{newPrivateStateEnc} & \triangleq \text{\textbf{SymmetricEncrypt}(LL2\text{NVRAM}.\text{symmetricKey}, sResult.\text{newPrivateState})} \\
\text{currentHistorySummary} & \triangleq [ \\
\text{anchor} & \mapsto \text{LL2\text{NVRAM}.\text{historySummaryAnchor},} \\
\text{extension} & \mapsto \text{LL2\text{SPCR}}] \\
\text{newHistorySummary} & \triangleq \text{Successor(\text{currentHistorySummary}, \text{input}, LL2\text{NVRAM}.\text{hashBarrier})} \\
\text{newHistorySummaryHash} & \triangleq \text{\text{Hash(\text{newHistorySummary}.\text{anchor}, newHistorySummary.\text{extension})}} \\
\text{newStateHash} & \triangleq \text{\text{Hash(sResult.\text{newPublicState}, \text{newPrivateStateEnc})}} \\
\text{newHistoryStateBinding} & \triangleq \text{\text{Hash(newHistorySummaryHash, newStateHash)}} \\
\text{newAuthenticator} & \triangleq \text{\textbf{GenerateMAC}(LL2\text{NVRAM}.\text{symmetricKey}, \text{newHistoryStateBinding})} \\
\end{align*}
\]

\[
\begin{align*}
\text{IN} \\
\wedge \text{historySummaryHash} \in \text{HashType} \\
\wedge \text{stateHash} \in \text{HashType} \\
\wedge \text{historyStateBinding} \in \text{HashType} \\
\wedge \text{privateState} \in \text{PrivateKeyType} \\
\wedge \text{sResult} \in \text{ServiceResultType} \\
\wedge \text{sResult.\text{newPublicState}} \in \text{PublicKeyType} \\
\wedge \text{sResult.\text{newPrivateState}} \in \text{PrivateKeyType} \\
\wedge \text{sResult.\text{output}} \in \text{OutputType} \\
\wedge \text{newPrivateStateEnc} \in \text{PrivateKeyEncType} \\
\wedge \text{currentHistorySummary} \in \text{HistorySummaryType} \\
\wedge \text{currentHistorySummary.\text{anchor}} \in \text{HashType} \\
\wedge \text{currentHistorySummary.\text{extension}} \in \text{HashType} \\
\wedge \text{newHistorySummary} \in \text{HistorySummaryType} \\
\wedge \text{newHistorySummary.\text{anchor}} \in \text{HashType} \\
\wedge \text{newHistorySummary.\text{extension}} \in \text{HashType} \\
\wedge \text{newHistorySummaryHash} \in \text{HashType} \\
\wedge \text{newStateHash} \in \text{HashType} \\
\wedge \text{newHistoryStateBinding} \in \text{HashType} \\
\wedge \text{newAuthenticator} \in \text{MACType} \\
\end{align*}
\]

\(1\). \text{\textbf{take}} \text{input} \in \text{LL2AvailableInputs} \\
(1) \text{\textbf{historySummaryHash} } \triangleq \text{\text{Hash(LL2\text{RAM}.\text{historySummary.\text{anchor}}, LL2\text{RAM}.\text{historySummary.\text{extension}})}} \\
(1) \text{\textbf{stateHash} } \triangleq \text{\text{Hash(LL2\text{RAM}.\text{publicState}, LL2\text{RAM}.\text{privateStateEnc})}} \\
(1) \text{\textbf{historyStateBinding} } \triangleq \text{\text{Hash(\text{historySummaryHash}, stateHash)}} \\
(1) \text{\textbf{privateState} } \triangleq \text{\text{SymmetricDecrypt(LL2\text{NVRAM}.\text{symmetricKey}, LL2\text{RAM}.\text{privateStateEnc})}} \\
(1) \text{\textbf{sResult} } \triangleq \text{\text{Service(LL2\text{RAM}.\text{publicState}, privateState, input)}} \\
(1) \text{\textbf{newPrivateStateEnc} } \triangleq \text{\text{\textbf{SymmetricEncrypt}(LL2\text{NVRAM}.\text{symmetricKey}, sResult.\text{newPrivateState})}} \\
(1) \text{\textbf{currentHistorySummary} } \triangleq [ \\
\text{anchor} & \mapsto \text{LL2\text{NVRAM}.\text{historySummaryAnchor},} \\
\text{extension} & \mapsto \text{LL2\text{SPCR}}] \\
(1) \text{\textbf{newHistorySummary} } \triangleq \text{\text{Successor(\text{currentHistorySummary}, \text{input}, LL2\text{NVRAM}.\text{hashBarrier})}} \\
(1) \text{\textbf{newStateHash} } \triangleq \text{\text{Hash(newHistorySummary.\text{anchor}, newHistorySummary.\text{extension})}} \\
(1) \text{\textbf{newHistoryStateBinding} } \triangleq \text{\text{Hash(newHistorySummaryHash, newStateHash)}} \\
(1) \text{\textbf{newAuthenticator} } \triangleq \text{\text{\textbf{GenerateMAC}(LL2\text{NVRAM}.\text{symmetricKey}, \text{newHistoryStateBinding})}} \\
(1) \text{\textbf{HIDE DEF} historySummaryHash, stateHash, historyStateBinding, privateState,}
\( sResult, \text{newPrivateStateEnc}, \text{currentHistorySummary}, \text{newHistorySummary}, \)
\( \text{newHistorySummaryHash}, \text{newStateHash}, \text{newHistoryStateBinding}, \text{newAuthenticator} \)

(1)2. HAVE LL2TypeInvariant

(1)3. historySummaryHash ∈ HashType

(2)1. LL2RAM.historySummary.anchor ∈ HashDomain

(3)1. LL2RAM.historySummary.anchor ∈ HashType

BY (1)2, LL2SubtypeImplicationLemma\( \text{Def} \) LL2SubtypeImplication

(3)2. QED

BY (3)1 \text{Def} HashDomain

(2)2. LL2RAM.historySummary.extension ∈ HashDomain

(3)1. LL2RAM.historySummary.extension ∈ HashType

BY (1)2, LL2SubtypeImplicationLemma\( \text{Def} \) LL2SubtypeImplication

(3)2. QED

BY (3)1 \text{Def} HashDomain

(2)3. QED

BY (2)1, (2)2, HashTypeSafe\( \text{Def} \) historySummaryHash

(1)4. stateHash ∈ HashType

(2)1. ∧ LL2RAM.publicState ∈ PublicStateType

∧ LL2RAM.privateStateEnc ∈ PrivateStateEncType

BY (1)2, LL2SubtypeImplicationLemma\( \text{Def} \) LL2SubtypeImplication

(2)2. ∧ LL2RAM.publicState ∈ HashDomain

∧ LL2RAM.privateStateEnc ∈ HashDomain

BY (2)1 \text{Def} HashDomain

(2)3. QED

BY (2)2, HashTypeSafe\( \text{Def} \) stateHash

(1)5. historyStateBinding ∈ HashType

(2)1. historySummaryHash ∈ HashDomain

BY (1)3 \text{Def} HashDomain

(2)2. stateHash ∈ HashDomain

BY (1)4 \text{Def} HashDomain

(2)3. QED

BY (2)1, (2)2, HashTypeSafe\( \text{Def} \) historyStateBinding

(1)6. privateState ∈ PrivateStateType

(2)1. ∧ LL2NVRAM.symmetricKey ∈ SymmetricKeyType

∧ LL2RAM.privateStateEnc ∈ PrivateStateEncType

BY (1)2, LL2SubtypeImplicationLemma\( \text{Def} \) LL2SubtypeImplication

(2)2. QED

BY (2)1, SymmetricDecryptionTypeSafe\( \text{Def} \) privateState

(1)7. sResult ∈ ServiceResultType

(2)1. LL2RAM.publicState ∈ PublicStateType

BY (1)2, LL2SubtypeImplicationLemma\( \text{Def} \) LL2SubtypeImplication

(2)2. privateState ∈ PrivateStateType

BY (1)6

(2)3. input ∈ InputType

(3)1. LL2AvailableInputs ⊆ InputType

BY (1)2 \text{Def} LL2TypeInvariant

(3)2. QED

BY (1)1, (3)1

(2)4. QED

BY (2)1, (2)2, (2)3, ServiceTypeSafe\( \text{Def} \) sResult

(1)8. ∧ sResult.newPublicState ∈ PublicStateType

∧ sResult.newPrivateState ∈ PrivateStateType
∧ sResult.output ∈ OutputType

BY (1) DEF ServiceResultType

(1).9. newPrivateStateEnc ∈ PrivateStateEncType

(2).1. LL2NVRAM.symmetricKey ∈ SymmetricKeyType
    BY (1)/2, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication

(2).2. sResult.newPrivateState ∈ PrivateStateType
    BY (1)/8

(2).3. QED
    BY (2).1, (2)/2, SymmetricEncryptionTypeSafe DEF newPrivateStateEnc

(1).10. currentHistorySummary ∈ HistorySummaryType

(2).1. LL2NVRAM.historySummaryAnchor ∈ HashType
    BY (1)/2, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication

(2).2. LL2SPCR ∈ HashType
    BY (1)/2 DEF LL2TypeInvariant

BY (2).1, (2)/2 DEF currentHistorySummary, HistorySummaryType

(1).11. ∧ currentHistorySummary.anchor ∈ HashType
    ∧ currentHistorySummary.extension ∈ HashType
    BY (1)/10 DEF HistorySummaryType

(1).12. newHistorySummary ∈ HistorySummaryType

(2).1. currentHistorySummary ∈ HistorySummaryType
    BY (1)/10

(2).2. input ∈ InputType
    (3).1. input ∈ LL2AvailableInputs
        BY (1)/1

(3).2. LL2AvailableInputs ⊆ InputType
        BY (1)/2 DEF LL2TypeInvariant

(3).3. QED
        BY (3).1, (3)/2

(2).3. LL2NVRAM.hashBarrier ∈ HashType
    BY (1)/2, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication

(2).4. QED
    BY (2).1, (2)/2, (2)/3, SuccessorTypeSafe DEF newHistorySummary

(1).13. ∧ newHistorySummary.anchor ∈ HashType
    ∧ newHistorySummary.extension ∈ HashType
    BY (1)/12 DEF HistorySummaryType

(1).14. newHistorySummaryHash ∈ HashType

(2).1. newHistorySummary anchors ∈ HashDomain
    (3).1. newHistorySummary.anchor ∈ HashType
        BY (1)/12 DEF HistorySummaryType

(3).2. QED
        BY (3).1 DEF HashDomain

(2).2. newHistorySummary.extension ∈ HashDomain
    (3).1. newHistorySummary.extension ∈ HashType
        BY (1)/12 DEF HistorySummaryType

(3).2. QED
        BY (3).1 DEF HashDomain

(2).3. QED
    BY (2).1, (2)/2, HashTypeSafe DEF newHistorySummaryHash

(1).15. newStateHash ∈ HashType

(2).1. sResult.newPublicState ∈ HashDomain
    (3).1. sResult.newPublicState ∈ PublicStateType
theorem: Let action all have the appropriate type. This is a trivial proof that merely walks through the definitions.

The `LL2RepeatOperationDefsTypeSafeLemma` proves that the definitions within the `LET` of the `LL2RepeatOperation` action all have the appropriate type. This is a trivial proof that merely walks through the definitions.

**Theorem:** `LL2RepeatOperationDefsTypeSafeLemma` \( \triangleq \)
\[ \forall \text{input} \in \text{LL2AvailableInputs} : \]
\[ \text{LL2TypeInvariant} \Rightarrow \]
\[ \text{LET} \]
\[ \text{historySummaryHash} \triangleq \]
\[ \text{Hash}(\text{LL2RAM.historySummary.anchor}, \text{LL2RAM.historySummary.extension}) \]
\[ \text{stateHash} \triangleq \text{Hash}(\text{LL2RAM.publicState}, \text{LL2RAM.privateStateEnc}) \]
\[ \text{historyStateBinding} \triangleq \text{Hash}(\text{historySummaryHash}, \text{stateHash}) \]
\[ \text{newHistorySummary} \triangleq \]
\[ \text{Successor}(\text{LL2RAM.historySummary}, \text{input}, \text{LL2NVRAM.hashBarrier}) \]
\[ \text{checkpointedHistorySummary} \triangleq \text{Checkpoint}(\text{LL2RAM.historySummary}) \]
\[ \text{newCheckpointedHistorySummary} \triangleq \]
\[ \text{Successor}(\text{checkpointedHistorySummary}, \text{input}, \text{LL2NVRAM.hashBarrier}) \]
\[ \text{checkpointedNewHistorySummary} \triangleq \text{Checkpoint}(\text{newHistorySummary}) \]
\[ \text{checkpointedNewCheckpointedHistorySummary} \triangleq \]
\[ \text{Checkpoint}(\text{newCheckpointedHistorySummary}) \]
\[ \text{privateState} \triangleq \text{SymmetricDecrypt}(\text{LL2NVRAM.symmetricKey}, \text{LL2RAM.privateStateEnc}) \]
\[ \text{sResult} \triangleq \text{Service}(\text{LL2RAM.publicState}, \text{privateState}, \text{input}) \]
\[ \text{newPrivateStateEnc} \triangleq \]
\[ \text{SymmetricEncrypt}(\text{LL2NVRAM.symmetricKey}, \text{sResult.newPrivateState}) \]
\[ \text{currentHistorySummary} \triangleq [\]
\[ \text{anchor} \mapsto \text{LL2NVRAM.historySummaryAnchor}, \]

```plaintext
BY (1)8
(3)2. QED
BY (3)1 DEF HashDomain
(2)2. newPrivateStateEnc ∈ HashDomain
BY (1)9 DEF HashDomain
(2)3. QED
BY (2)1, (2)2, HashTypeSafeDEF newStateHash
(1)16. newHistoryStateBinding ∈ HashType
(2)1. newHistorySummaryHash ∈ HashDomain
BY (1)14 DEF HashDomain
(2)2. newStateHash ∈ HashDomain
BY (1)15 DEF HashDomain
(2)3. QED
BY (2)1, (2)2, HashTypeSafeDEF newHistoryStateBinding
(1)17. newAuthenticator ∈ MACType
(2)1. LL2NVRAM.symmetricKey ∈ SymmetricKeyType
BY (1)2, LL2SubtypeImplicationLemmaDEF LL2SubtypeImplication
(2)2. newHistoryStateBinding ∈ HashType
BY (1)16
(2)3. QED
BY (2)1, (2)2, GenerateMACTypeSafeDef newAuthenticator
(1)18. QED.
BY (1)3, (1)4, (1)5, (1)6, (1)7, (1)8, (1)9,
(1)10, (1)11, (1)12, (1)13, (1)14, (1)15, (1)16, (1)17
DEF historySummaryHash, stateHash, historyStateBinding, privateState,
sResult, newPrivateStateEnc, currentHistorySummary, newHistorySummary,
newHistorySummaryHash, newStateHash, newHistoryStateBinding, newAuthenticator
```

204
extension ↦ LL2SPCR[ ]
currentHistorySummaryHash ≜ Hash(LL2NVRAM.historySummaryAnchor, LL2SPCR)
newStateHash ≜ Hash(sResult, newPublicState, newPrivateStateEnc)
newHistoryStateBinding ≜ Hash(currentHistorySummaryHash, newStateHash)
newAuthenticator ≜ GenerateMAC(LL2NVRAM.symmetricKey, newHistoryStateBinding)

\[\begin{aligned}
&\land \text{historySummaryHash} \in \text{HashType} \\
&\land \text{stateHash} \in \text{HashType} \\
&\land \text{historyStateBinding} \in \text{HashType} \\
&\land \text{newHistorySummary} \in \text{HistorySummaryType} \\
&\land \text{newHistorySummary.} \text{anchor} \in \text{HashType} \\
&\land \text{newHistorySummary.} \text{extension} \in \text{HashType} \\
&\land \text{checkpointedHistorySummary} \in \text{HistorySummaryType} \\
&\land \text{checkpointedHistorySummary.} \text{anchor} \in \text{HashType} \\
&\land \text{checkpointedHistorySummary.} \text{extension} \in \text{HashType} \\
&\land \text{checkpointedNewHistorySummary} \in \text{HistorySummaryType} \\
&\land \text{checkpointedNewHistorySummary.} \text{anchor} \in \text{HashType} \\
&\land \text{checkpointedNewHistorySummary.} \text{extension} \in \text{HashType} \\
&\land \text{checkpointedNewHistorySummary.} \text{extension} \in \text{HashType} \\
&\land \text{checkpointedNewCheckpointedHistorySummary} \in \text{HistorySummaryType} \\
&\land \text{checkpointedNewCheckpointedHistorySummary.} \text{anchor} \in \text{HashType} \\
&\land \text{checkpointedNewCheckpointedHistorySummary.} \text{extension} \in \text{HashType} \\
&\land \text{privateState} \in \text{PrivateStateType} \\
&\land \text{sResult} \in \text{ServiceResultType} \\
&\land \text{sResult.} \text{newPublicState} \in \text{PublicStateType} \\
&\land \text{sResult.} \text{newPrivateKey} \in \text{PrivateStateType} \\
&\land \text{sResult.} \text{output} \in \text{OutputType} \\
&\land \text{newPrivateKey} \in \text{PrivateKeyType} \\
&\land \text{currentHistorySummary} \in \text{HistorySummaryType} \\
&\land \text{currentHistorySummary.} \text{anchor} \in \text{HashType} \\
&\land \text{currentHistorySummary.} \text{extension} \in \text{HashType} \\
&\land \text{currentHistorySummaryHash} \in \text{HashType} \\
&\land \text{newStateHash} \in \text{HashType} \\
&\land \text{newHistoryStateBinding} \in \text{HashType} \\
&\land \text{newAuthenticator} \in \text{MACType} \\
\end{aligned}\]

\(1\) Take input in LL2AvailableInputs

\(1\) historySummaryHash ≜ \\
\(\text{Hash(LL2RAM.} \text{historySummaryAnchor, LL2RAM.} \text{historySummary.extension})\)

(1) stateHash ≜ \\
\(\text{Hash(LL2RAM.} \text{publicState, LL2RAM.} \text{privateStateEnc})\)

(1) historyStateBinding ≜ \\
\(\text{Hash(historySummaryHash, stateHash})\)

(1) newHistorySummary ≜ \\
\(\text{Successor(LL2RAM.} \text{historySummary, input, LL2NVRAM.hashBarrier})\)

(1) checkpointedHistorySummary ≜ \\
\(\text{Checkpoint(LL2RAM.} \text{historySummary})\)

(1) newCheckpointedHistorySummary ≜ \\
\(\text{Successor(checkpointedHistorySummary, input, LL2NVRAM.hashBarrier})\)

(1) checkpointedNewHistorySummary ≜ \\
\(\text{Checkpoint(newHistorySummary})\)

(1) checkpointedNewCheckpointedHistorySummary ≜ \\
\(\text{Checkpoint(newCheckpointedHistorySummary})\)

(1) privateKey ≜ \\
\(\text{SymmetricDecrypt(LL2NVRAM.} \text{symmetricKey, LL2RAM.} \text{privateStateEnc})\)

(1) sResult ≜ \\
\(\text{Service(LL2RAM.} \text{publicState, privateState, input})\)

(1) newPrivateKey ≜ \\
\(\text{LL2NVRAM.historySummaryAnchor, LL2RAM.historySummaryextension})\)

(1) stateHash ≜ \\
\(\text{Hash(LL2RAM.} \text{publicState, LL2RAM.} \text{privateStateEnc})\)

(1) historyStateBinding ≜ \\
\(\text{Hash(historySummaryHash, stateHash})\)

(1) newHistorySummary ≜ \\
\(\text{Successor(LL2RAM.} \text{historySummary, input, LL2NVRAM.hashBarrier})\)

(1) checkpointedHistorySummary ≜ \\
\(\text{Checkpoint(LL2RAM.} \text{historySummary})\)

(1) newCheckpointedHistorySummary ≜ \\
\(\text{Successor(checkpointedHistorySummary, input, LL2NVRAM.hashBarrier})\)

(1) checkpointedNewHistorySummary ≜ \\
\(\text{Checkpoint(newHistorySummary})\)

(1) checkpointedNewCheckpointedHistorySummary ≜ \\
\(\text{Checkpoint(newCheckpointedHistorySummary})\)

(1) privateKey ≜ \\
\(\text{SymmetricDecrypt(LL2NVRAM.} \text{symmetricKey, LL2RAM.} \text{privateStateEnc})\)

(1) sResult ≜ \\
\(\text{Service(LL2RAM.} \text{publicState, privateState, input})\)

(1) newPrivateKey ≜ \

205
SymmetricEncrypt(LL2NVRAM.symmetricKey, sResult.newPrivateState)

1. currentHistorySummary ≜ [anchor ↦ LL2NVRAM.historySummaryAnchor, extension ↦ LL2SPCR]

1. currentHistorySummaryHash ≜ Hash(LL2NVRAM.historySummaryAnchor, LL2SPCR)

1. newStateHash ≜ Hash(sResult.newPublicState, newPrivateStateEnc)

1. newHistoryStateBinding ≜ Hash(currentHistorySummaryHash, newStateHash)

1. newAuthenticator ≜ GenerateMAC(LL2NVRAM.symmetricKey, newHistoryStateBinding)

1. HIDE DEF historySummaryHash, stateHash, historyStateBinding, newHistorySummary, checkpointedHistorySummary, checkpointedNewHistorySummary, privateState, sResult, newPrivateStateEnc, currentHistorySummary, currentHistorySummaryHash, newStateHash, newHistoryStateBinding, newAuthenticator

1/2. HAVE LL2TypeInvariant

1/3. input ∈ InputType

(2.1) input ∈ LL2AvailableInputs

BY (1)1

(2.2) LL2AvailableInputs ⊆ InputType

BY (1)2 DEF LL2TypeInvariant

(2.3) QED

BY (2)1, (2)2

1/4. LL2NVRAM.hashBarrier ∈ HashType

BY (1)2, LL2SubtypeImplicationLemmaDEF LL2SubtypeImplication

1/5. historySummaryHash ∈ HashType

(2.1) LL2RAM.historySummary.anchor ∈ HashDomain

(3.1) LL2RAM.historySummary.anchor ∈ HashType

BY (1)2, LL2SubtypeImplicationLemmaDEF LL2SubtypeImplication

(3.2) QED

BY (3)1 DEF HashDomain

(2.2) LL2RAM.historySummary.extension ∈ HashDomain

(3.1) LL2RAM.historySummary.extension ∈ HashType

BY (1)2, LL2SubtypeImplicationLemmaDEF LL2SubtypeImplication

(3.2) QED

BY (3)1 DEF HashDomain

(2.3) QED

BY (2)1, (2)2, HashTypeSafeDEF historySummaryHash

1/6. stateHash ∈ HashType

(2.1) ∧ LL2RAM.publicState ∈ PublicStateType

∧ LL2RAM.privateStateEnc ∈ PrivateStateEncType

BY (1)2, LL2SubtypeImplicationLemmaDEF LL2SubtypeImplication

(2.2) ∧ LL2RAM.publicState ∈ HashDomain

∧ LL2RAM.privateStateEnc ∈ HashDomain

BY (2)1 DEF HashDomain

(2.3) QED

BY (2)2, HashTypeSafeDEF stateHash

1/7. historyStateBinding ∈ HashType

(2.1) historySummaryHash ∈ HashDomain

BY (1)5 DEF HashDomain

(2.2) stateHash ∈ HashDomain

BY (1)6 DEF HashDomain

(2.3) QED
BY (2)1, (2)2, HashTypeSafeDef historyStateBinding

(1)8. newHistorySummary ∈ HistorySummaryType

(2)1. LL2RAM.historySummary ∈ HistorySummaryType
   BY (1)2, LL2SubtypeImplicationLemmaDef LL2SubtypeImplication
(2)2. QED
   BY (1)3, (1)4, (2)1, SuccessorTypeSafeDef newHistorySummary
(1)9. ∧ newHistorySummary.anchor ∈ HashType
   ∧ newHistorySummary.extension ∈ HashType
   BY (1)8 DEF HistorySummaryType
(1)10. checkpointedHistorySummary ∈ HistorySummaryType
       BY (1)2, LL2SubtypeImplicationLemmaDef LL2SubtypeImplication
       BY (1)9, SuccessorTypeSafeDef checkpointedHistorySummary
(1)11. ∧ checkpointedHistorySummary.anchor ∈ HashType
       ∧ checkpointedHistorySummary.extension ∈ HashType
       BY (1)10 DEF HistorySummaryType
(1)12. newCheckpointedHistorySummary ∈ HistorySummaryType
       BY (1)3, (1)4, (1)10, SuccessorTypeSafeDef newCheckpointedHistorySummary
(1)13. ∧ newCheckpointedHistorySummary.anchor ∈ HashType
       ∧ newCheckpointedHistorySummary.extension ∈ HashType
       BY (1)12 DEF HistorySummaryType
(1)14. checkpointedNewHistorySummary ∈ HistorySummaryType
       BY (1)8, CheckpointTypeSafeDef checkpointedNewHistorySummary
(1)15. ∧ checkpointedNewHistorySummary.anchor ∈ HashType
       ∧ checkpointedNewHistorySummary.extension ∈ HashType
       BY (1)14 DEF HistorySummaryType
(1)16. checkpointedNewCheckpointedHistorySummary ∈ HistorySummaryType
       BY (1)12, CheckpointTypeSafeDef checkpointedNewCheckpointedHistorySummary
(1)17. ∧ checkpointedNewCheckpointedHistorySummary.anchor ∈ HashType
       ∧ checkpointedNewCheckpointedHistorySummary.extension ∈ HashType
       BY (1)16 DEF HistorySummaryType
(1)18. privateState ∈ PrivateStateType
   (2)1. ∧ LL2NVRAM.symmetricKey ∈ SymmetricKeyType
       ∧ LL2RAM.privateStateEnc ∈ PrivateStateEncType
   BY (1)2, LL2SubtypeImplicationLemmaDef LL2SubtypeImplication
   (2)2. QED
   BY (2)1, SymmetricDecryptionTypeSafeDef privateState
(1)19. sResult ∈ ServiceResultType
   (2)1. LL2RAM.publicState ∈ PublicStateType
       BY (1)2, LL2SubtypeImplicationLemmaDef LL2SubtypeImplication
   (2)2. privateState ∈ PrivateStateType
       BY (1)18
   (2)3. input ∈ InputType
       (3)1. LL2AvailableInputs ⊆ InputType
           BY (1)2 DEF LL2TypeInvariant
       (3)2. QED
           BY (1)1, (3)1
   (2)4. QED
   BY (2)1, (2)2, (2)3, ServiceTypeSafeDef sResult
(1)20. ∧ sResult.newPublicState ∈ PublicStateType
       ∧ sResult.newPrivateState ∈ PrivateStateType
\( sResult.output \in OutputType \)

by (1)19 def ServiceResultType

(1)21. \( newPrivateStateEnc \in PrivateStateEncType \)

(2)1. \( LL2NVRAM\textunderscore symmetricKey \in SymmetricKeyType \)

by (1)2, LL2SubtypeImplicationLemma def LL2SubtypeImplication

(2)2. \( sResult.newPrivateState \in PrivateStateType \)

by (1)20

(2)3. QED

by (2)1, (2)2, SymmetricEncryptionTypeSafe def newPrivateStateEnc

(1)22. \( currentHistorySummary \in HistorySummaryType \)

(2)1. \( LL2NVRAM\textunderscore historySummaryAnchor \in HashType \)

by (1)2, LL2SubtypeImplicationLemma def LL2SubtypeImplication

(2)2. \( LL2SPCR \in HashType \)

by (1)2 def LL2TypeInvariant

(2)3. QED

by (2)1, (2)2 def currentHistorySummary, HistorySummaryType

(1)23. \( \land currentHistorySummary\textunderscore anchor \in HashType \)

\( \land currentHistorySummary\textunderscore extension \in HashType \)

by (1)22 def HistorySummaryType

(1)24. \( currentHistorySummaryHash \in HashType \)

(2)1. \( LL2NVRAM\textunderscore historySummaryAnchor \in HashDomain \)

(3)1. \( LL2NVRAM\textunderscore historySummaryAnchor \in HashType \)

by (1)2, LL2SubtypeImplicationLemma def LL2SubtypeImplication

(3)2. QED

by (3)1 def HashDomain

(2)2. \( LL2SPCR \in HashDomain \)

(3)1. \( LL2SPCR \in HashType \)

by (1)2 def LL2TypeInvariant

(3)2. QED

by (3)1 def HashDomain

(2)3. QED

by (2)1, (2)2, HashTypeSafe def currentHistorySummaryHash

(1)25. \( newStateHash \in HashType \)

(2)1. \( sResult.newPublicState \in HashDomain \)

(3)1. \( sResult.newPublicState \in PublicStateType \)

by (1)20

(3)2. QED

by (3)1 def HashDomain

(2)2. \( newPrivateStateEnc \in HashDomain \)

by (1)21 def HashDomain

(2)3. QED

by (2)1, (2)2, HashTypeSafe def newStateHash

(1)26. \( newHistoryStateBinding \in HashType \)

(2)1. \( currentHistorySummaryHash \in HashDomain \)

by (1)24 def HashDomain

(2)2. \( newStateHash \in HashDomain \)

by (1)25 def HashDomain

(2)3. QED

by (2)1, (2)2, HashTypeSafe def newHistoryStateBinding

(1)27. \( newAuthenticator \in MACType \)

(2)1. \( LL2NVRAM\textunderscore symmetricKey \in SymmetricKeyType \)

by (1)2, LL2SubtypeImplicationLemma def LL2SubtypeImplication
(2.2) \( \text{newHistoryStateBinding} \in \text{HashType} \)

BY (1.26)

(2.3) QED

BY (2.1), (2.2), \( \text{GenerateMACTypeSafe} \)

DEF newAuthenticator

The \( \text{LL2TakeCheckpointDefsTypeSafeLemma} \) proves that the definition within the \text{LET} of the \( \text{LL2TakeCheckpoint} \) action has the appropriate type. This is a trivial proof that merely walks through the definitions.

THEOREM \( \text{LL2TakeCheckpointDefsTypeSafeLemma} \triangleq \)

\( \text{LL2TypeInvariant} \Rightarrow \)

LET

\( \text{newHistorySummaryAnchor} \triangleq \text{Hash(\text{LL2NVRAM}.\text{historySummaryAnchor}, \text{LL2SPCR})} \)

IN

\( \text{newHistorySummaryAnchor} \in \text{HashType} \)

(1) \( \text{newHistorySummaryAnchor} \triangleq \text{Hash(\text{LL2NVRAM}.\text{historySummaryAnchor}, \text{LL2SPCR})} \)

(1) HAVE \( \text{LL2TypeInvariant} \)

(1.2) \( \text{newHistorySummaryAnchor} \in \text{HashType} \)

(2.1) \( \text{LL2NVRAM}.\text{historySummaryAnchor} \in \text{HashDomain} \)

(3.1) \( \text{LL2NVRAM}.\text{historySummaryAnchor} \in \text{HashType} \)

BY (1.1), \( \text{LL2SubtypeImplicationLemma} \)

DEF \( \text{LL2SubtypeImplicationLemma} \)

(3.2) QED

BY (3.1) DEF \( \text{HashDomain} \)

(2.2) \( \text{LL2SPCR} \in \text{HashDomain} \)

(3.1) \( \text{LL2SPCR} \in \text{HashType} \)

BY (1.1) DEF \( \text{LL2TypeInvariant} \)

(3.2) QED

BY (3.1) DEF \( \text{HashDomain} \)

(2.3) QED

BY (2.1), (2.2), \( \text{HashTypeSafe} \)

DEF newHistorySummaryAnchor

The \( \text{LL2CorruptSPCRDefsTypeSafeLemma} \) proves that the definition within the \text{LET} of the \( \text{LL2CorruptSPCR} \) action has the appropriate type. This is a trivial proof that merely walks through the definitions.

THEOREM \( \text{LL2CorruptSPCRDefsTypeSafeLemma} \triangleq \)

\( \forall \text{fakeHash} \in \text{HashDomain} : \)

\( \text{LL2TypeInvariant} \Rightarrow \)

LET

\( \text{newHistorySummaryExtension} \triangleq \text{Hash(\text{LL2SPCR}, \text{fakeHash})} \)
The AuthenticatorsMatchDefsTypeSafeLemma proves that the definitions within the LET of the AuthenticatorsMatch predicate have the appropriate type. This is a trivial proof that merely walks through the definitions.
The TypeSafetyRefinementLemma states that if the Memoir-Opt type invariant holds, and the refinement holds, then the Memoir-Basic type invariant holds.

THEOREM TypeSafetyRefinementLemma \( \triangleq \)

\( LL_2 \text{TypeInvariant} \land LL_2 \text{Refinement} \Rightarrow LL_1 \text{TypeInvariant} \)

\( \langle 1 \rangle 1. \text{HAVE} \ LL_2 \text{TypeInvariant} \land LL_2 \text{Refinement} \)

\( \langle 1 \rangle 2. \ LL_1 \text{AvailableInputs} \subseteq \text{InputType} \)

\( \langle 2 \rangle 1. \ LL_1 \text{AvailableInputs} = LL_2 \text{AvailableInputs} \)

\( \langle 1 \rangle 3. \text{DEF} LL_1 \text{Refinement} \)

\( \langle 2 \rangle 2. \ LL_1 \text{AvailableInputs} \subseteq \text{InputType} \)

\( \langle 2 \rangle 3. \text{QED} \)

\( \langle 1 \rangle 3. \ LL_1 \text{ObservedOutputs} \subseteq \text{OutputType} \)

\( \langle 2 \rangle 1. \ LL_1 \text{ObservedOutputs} = LL_2 \text{ObservedOutputs} \)

\( \langle 2 \rangle 2. \text{DEF} LL_2 \text{Refinement} \)

\( 211 \)
(2)2. $LL2ObservedOutputs \subseteq OutputType$
   BY (1)1 DEF $LL2TypeInvariant$
(2)3. QED
   BY (2)1, (2)2
(1)4. $LL1ObservedAuthenticators \subseteq MACType$
   BY (1)1 DEF $LL2Refinement$
(1)5. $LL1Disk \in LL1UntrustedStorageType$
   BY (1)1 DEF $LL2Refinement$
(1)6. $LL1RAM \in LL1UntrustedStorageType$
   BY (1)1 DEF $LL2Refinement$
(1)7. $LL1NVRAM \in LL1TrustedStorageType$
   BY (1)1 DEF $LL2Refinement$
(1)8. QED
   BY (1)2, (1)3, (1)4, (1)5, (1)6, (1)7 DEF $LL1TypeInvariant$
4.9 Proof of Type Safety of the Memoir-Opt Spec

This module proves the type safety of the Memoir-Opt spec.

EXTENDS MemoirLL2TypeLemmas

THEOREM LL2TypeSafe \iff LL2Spec \Rightarrow \Box LL2TypeInvariant

The top level of the proof is boilerplate TLA+ for an Inv1-style proof. First, we prove that the initial state satisfies LL2TypeInvariant. Second, we prove that the LL2Next predicate inductively preserves LL2TypeInvariant. Third, we use temporal induction to prove that these two conditions satisfy type safety over all behaviors.

⟨1⟩1. LL2Init ⇒ LL2TypeInvariant

The base case follows directly from the definition of LL2Init. There are a bunch of steps, but they are simple expansions of definitions and appeals to the type safety of the initial definitions.

⟨2⟩1. HAVE LL2Init

⟨2⟩2. PICK symmetricKey ∈ SymmetricKeyType, hashBarrier ∈ HashType :

   LL2Init!(symmetricKey, hashBarrier)!1

BY ⟨2⟩1 \def LL2Init

⟨2⟩3. initialPrivateStateEnc ≜ SymmetricEncrypt(symmetricKey, InitialPrivateState)
⟨2⟩4. initialStateHash ≜ Hash(InitialPublicState, initialPrivateStateEnc)
⟨2⟩5. initialHistorySummary ≜ [
   anchor ↦ BaseHashValue,
   extension ↦ BaseHashValue
]
⟨2⟩6. initialHistorySummaryHash ≜ Hash(BaseHashValue, BaseHashValue)
⟨2⟩7. initialHistoryStateBinding ≜ Hash(initialHistorySummaryHash, initialStateHash)
⟨2⟩8. initialAuthenticator ≜ GenerateMAC(symmetricKey, initialHistoryStateBinding)
⟨2⟩9. initialUntrustedStorage ≜ [
   publicState ↦ InitialPublicState,
   privateStateEnc ↦ initialPrivateStateEnc,
   historySummary ↦ initialHistorySummary,
   authenticator ↦ initialAuthenticator
]
⟨2⟩10. initialTrustedStorage ≜ [
   historySummaryAnchor ↦ BaseHashValue,
   symmetricKey ↦ symmetricKey,
   hashBarrier ↦ hashBarrier,
   extensionInProgress ↦ FALSE
]

⟨3⟩1. symmetricKey ∈ SymmetricKeyType
⟨3⟩2. QED
BY ⟨3⟩1, LL2InitDefsTypeSafeLemma

⟨2⟩ HIDE DEF initialPrivateStateEnc, initialStateHash, initialHistorySummary,
    initialHistorySummaryHash, initialHistoryStateBinding, initialAuthenticator,
    initialUntrustedStorage, initialTrustedStorage

⟨2⟩4. LL2AvailableInputs ⊆ InputType
⟨3⟩1. LL2AvailableInputs = InitialAvailableInputs
BY (2)2
(3)2. InitialAvailableInputs ⊆ InputType
   BY ConstantsTypeSafe DEF ConstantsTypeSafe
(3)3. QED
   BY (3)1, (3)2

(2)5. LL2_ObservedOutputs ⊆ OutputType
(3)1. LL2_ObservedOutputs = {}
   BY (2)2
(3)2. QED
   BY (3)1

(2)6. LL2_ObservedAuthenticators ⊆ MACType
(3)1. LL2_ObservedAuthenticators = {initialAuthenticator}
   BY (2)2
   DEF initialAuthenticator, initialHistoryStateBinding, initialHistorySummaryHash,
      initialStateHash, initialPrivateStateEnc
(3)2. initialAuthenticator ∈ MACType
   BY (2)3
(3)3. QED
   BY (3)1, (3)2

(2)7. LL2_Disk ∈ LL2_UntrustedStorageType
(3)1. LL2_Disk = initialUntrustedStorage
   BY (2)2
   DEF initialUntrustedStorage, initialHistorySummary, initialAuthenticator,
      initialHistorySummaryHash, initialHistoryStateBinding, initialStateHash, initialPrivateStateEnc
(3)2. initialUntrustedStorage ∈ LL2_UntrustedStorageType
   BY (2)3
(3)3. QED
   BY (3)1, (3)2

(2)8. LL2_RAM ∈ LL2_UntrustedStorageType
(3)1. LL2_RAM = initialUntrustedStorage
   BY (2)2
   DEF initialUntrustedStorage, initialHistorySummary, initialAuthenticator,
      initialHistorySummaryHash, initialHistoryStateBinding, initialStateHash, initialPrivateStateEnc
(3)2. initialUntrustedStorage ∈ LL2_UntrustedStorageType
   BY (2)3
(3)3. QED
   BY (3)1, (3)2

(2)9. LL2_NVRAM ∈ LL2_TrustedStorageType
(3)1. LL2_NVRAM = initialTrustedStorage
   BY (2)2 DEF initialTrustedStorage
(3)2. initialTrustedStorage ∈ LL2_TrustedStorageType
   BY (2)3
(3)3. QED
   BY (3)1, (3)2

(2)10. LL2_SPCR ∈ HashType
(3)1. LL2_SPCR = BaseHashValue
   BY (2)2
(3)2. BaseHashValue ∈ HashType
   BY BaseHashValueTypeSafe
(3)3. QED
   BY (3)1, (3)2

(2)11. QED
The induction step is also straightforward. We assume the antecedents of the implication, then show that the consequent holds for all nine $LL2_{Next}$ actions plus stuttering.

(2.1. HAVE $LL2_{TypeInvariant} \land [LL2_{Next}]_{LL2_{Vars}}$

(2.2. CASE UNCHANGED $LL2_{Vars}$

Type safety is inductively trivial for a stuttering step.

(3.1. $LL2_{AvailableInputs}' \subseteq InputType$
(4.1. $LL2_{AvailableInputs} \subseteq InputType$
  BY (2.1) DEF $LL2_{TypeInvariant}$
(4.2. UNCHANGED $LL2_{AvailableInputs}$
  BY (2.2) DEF $LL2_{Vars}$
(4.3. QED
  BY (4.1), (4.2)

(3.2. $LL2_{ObservedOutputs}' \subseteq OutputType$
(4.1. $LL2_{ObservedOutputs} \subseteq OutputType$
  BY (2.1) DEF $LL2_{TypeInvariant}$
(4.2. UNCHANGED $LL2_{ObservedOutputs}$
  BY (2.2) DEF $LL2_{Vars}$
(4.3. QED
  BY (4.1), (4.2)

(3.3. $LL2_{ObservedAuthenticators}' \subseteq MACType$
(4.1. $LL2_{ObservedAuthenticators} \subseteq MACType$
  BY (2.1) DEF $LL2_{TypeInvariant}$
(4.2. UNCHANGED $LL2_{ObservedAuthenticators}$
  BY (2.2) DEF $LL2_{Vars}$
(4.3. QED
  BY (4.1), (4.2)

(3.4. $LL2_{Disk}' \in LL2_{UntrustedStorageType}$
(4.1. $LL2_{Disk} \in LL2_{UntrustedStorageType}$
  BY (2.1) DEF $LL2_{TypeInvariant}$
(4.2. UNCHANGED $LL2_{Disk}$
  BY (2.2) DEF $LL2_{Vars}$
(4.3. QED
  BY (4.1), (4.2)

(3.5. $LL2_{RAM}' \in LL2_{UntrustedStorageType}$
(4.1. $LL2_{RAM} \in LL2_{UntrustedStorageType}$
  BY (2.1) DEF $LL2_{TypeInvariant}$
(4.2. UNCHANGED $LL2_{RAM}$
  BY (2.2) DEF $LL2_{Vars}$
(4.3. QED
  BY (4.1), (4.2)

(3.6. $LL2_{NVRAM}' \in LL2_{TrustedStorageType}$
(4.1. $LL2_{NVRAM} \in LL2_{TrustedStorageType}$
  BY (2.1) DEF $LL2_{TypeInvariant}$
(4.2. UNCHANGED $LL2_{NVRAM}$
  BY (2.2) DEF $LL2_{Vars}$
(4.3. QED
  BY (4.1), (4.2)

(3.7. $LL2_{SPCR}' \in HashType$
(4.1. $LL2_{SPCR} \in HashType$
  BY (2.1) DEF $LL2_{TypeInvariant}$
Type safety is straightforward for a $LL2\text{MakeInputAvailable}$ action.

(4.1) PICK $\text{input} \in \text{InputType}$ : $LL2\text{MakeInputAvailable}!(\text{input})$

(4.2) $LL2\text{AvailableInputs}' \subseteq \text{InputType}$

(5.1) $LL2\text{AvailableInputs} \subseteq \text{InputType}$

(5.2) $LL2\text{AvailableInputs}' = LL2\text{AvailableInputs} \cup \{\text{input}\}$

(5.3) $\text{input} \in \text{InputType}$

(5.4) QED

(4.3) $LL2\text{ObservedOutputs}' \subseteq \text{OutputType}$

(5.1) $LL2\text{ObservedOutputs} \subseteq \text{OutputType}$

(5.2) UNCHANGED $LL2\text{ObservedOutputs}$

(5.3) QED

(4.4) $LL2\text{ObservedAuthenticators}' \subseteq \text{MACType}$

(5.1) $LL2\text{ObservedAuthenticators} \subseteq \text{MACType}$

(5.2) UNCHANGED $LL2\text{ObservedAuthenticators}$

(5.3) QED

(4.5) $LL2\text{Disk}' \in LL2\text{UntrustedStorageType}$

(5.1) $LL2\text{Disk} \in LL2\text{UntrustedStorageType}$

(5.2) UNCHANGED $LL2\text{Disk}$

(5.3) QED

(4.6) $LL2\text{RAM}' \in LL2\text{UntrustedStorageType}$

(5.1) $LL2\text{RAM} \in LL2\text{UntrustedStorageType}$

(5.2) UNCHANGED $LL2\text{RAM}$

(5.3) QED

(4.7) $LL2\text{NVRAM}' \in LL2\text{TrustedStorageType}$

(5.1) $LL2\text{NVRAM} \in LL2\text{TrustedStorageType}$

(5.2) UNCHANGED $LL2\text{NVRAM}$
BY \(4\)1
(5)3. QED
BY \(5\)1, \(5\)2
\(4\)8. LL2SPCR' \(\in\) HashType
(5)1. LL2SPCR \(\in\) HashType
BY \(2\)1 DEF LL2TypeInvariant
(5)2. UNCHANGED LL2SPCR
BY \(4\)1
(5)3. QED
BY \(5\)1, \(5\)2
\(4\)9. QED
BY \(4\)2, \(4\)3, \(4\)4, \(4\)5, \(4\)6, \(4\)7, \(4\)8 DEF LL2TypeInvariant

(3)2. CASE LL2PerformOperation

For a LL2PerformOperation action, we just walk through the definitions. Type safety follows directly.

\(4\)1. PICK input \(\in\) LL2AvailableInputs : LL2PerformOperation!(input)!1
BY \(3\)2 DEF LL2PerformOperation
\(4\) historySummaryHash \(\triangleq\)
\(\text{Hash}(\text{LL2RAM.historySummary.anchor}, \text{LL2RAM.historySummary.extension})\)
\(4\) stateHash \(\triangleq\) \(\text{Hash}(\text{LL2RAM.publicState}, \text{LL2RAM.privateStateEnc})\)
\(4\) historyStateBinding \(\triangleq\) \(\text{Hash}(\text{historySummaryHash}, \text{stateHash})\)
\(4\) privateState \(\triangleq\) SymmetricDecrypt(\(\text{LL2NVRAM.symmetricKey}, \text{LL2RAM.privateStateEnc}\))
\(4\) sResult \(\triangleq\) Service(\(\text{LL2RAM.publicState}, \text{privateState}, \text{input}\))
\(4\) newPrivateStateEnc \(\triangleq\)
\(\text{SymmetricEncrypt}(\text{LL2NVRAM.symmetricKey}, \text{sResult.newPrivateState})\)
\(4\) currentHistorySummary \(\triangleq\)
\(\left[\begin{array}{c}
\text{anchor} \rightarrow \text{LL2NVRAM.historySummaryAnchor}, \\
\text{extension} \rightarrow \text{LL2SPCR}
\end{array}\right]\)
\(4\) newHistorySummary \(\triangleq\) Successor(\(\text{currentHistorySummary}, \text{input}, \text{LL2NVRAM.hashBarrier}\))
\(4\) newHistorySummaryHash \(\triangleq\) \(\text{Hash}(\text{newHistorySummary.anchor}, \text{newHistorySummary.extension})\)
\(4\) newStateHash \(\triangleq\) \(\text{Hash}(\text{sResult.newPublicState}, \text{newPrivateStateEnc})\)
\(4\) newHistoryStateBinding \(\triangleq\) \(\text{Hash}(\text{newHistorySummaryHash}, \text{newStateHash})\)
\(4\) newAuthenticator \(\triangleq\) GenerateMAC(\(\text{LL2NVRAM.symmetricKey}, \text{newHistoryStateBinding}\))
\(4\)2. \(\land\)
\(\text{historySummaryHash} \in\) HashType
\(\land\) stateHash \(\in\) HashType
\(\land\) historyStateBinding \(\in\) HashType
\(\land\) privateState \(\in\) PrivateStateType
\(\land\) sResult \(\in\) ServiceResultType
\(\land\) sResult.newPublicState \(\in\) PublicStateType
\(\land\) sResult.newPrivateState \(\in\) PrivateStateType
\(\land\) sResult.output \(\in\) OutputType
\(\land\) newPrivateStateEnc \(\in\) PrivateStateEncType
\(\land\) currentHistorySummary \(\in\) HistorySummaryType
\(\land\) currentHistorySummary.anchor \(\in\) HashType
\(\land\) currentHistorySummary.extension \(\in\) HashType
\(\land\) newHistorySummary \(\in\) HistorySummaryType
\(\land\) newHistorySummary.anchor \(\in\) HashType
\(\land\) newHistorySummary.extension \(\in\) HashType
\(\land\) newHistorySummaryHash \(\in\) HashType
\(\land\) newStateHash \(\in\) HashType
\(\land\) newHistoryStateBinding \(\in\) HashType
\(\land\) newAuthenticator \(\in\) MACType
(5)1. input \(\in\) LL2AvailableInputs
BY (4)1
(5), 2. LL2TypeInvariant
BY (2)1
(5). QED
BY (5)1, (5)2. LL2PerformOperationDefsTypeSafeLemma
(4) HIDE DEF historySummaryHash, stateHash, historyStateBinding, privateState,
sResult, newPrivateStateEnc, currentHistorySummary, newHistorySummary,
newHistorySummaryHash, newStateHash, newHistoryStateBinding, newAuthenticator
(4)3. LL2AvailableInputs' ⊆ InputType
(5)1. LL2AvailableInputs ⊆ InputType
BY (2)1 DEF LL2TypeInvariant
(5)2. UNCHANGED LL2AvailableInputs
BY (4)1
(5)3. QED
BY (5)1, (5)2
(4)4. LL2ObservedOutputs' ⊆ OutputType
(5)1. LL2ObservedOutputs ⊆ OutputType
BY (2)1 DEF LL2TypeInvariant
(5)2. LL2ObservedOutputs' = LL2ObservedOutputs ∪ {sResult.output}
BY (4)1 DEF sResult, privateState
(5)3. sResult.output ∈ OutputType
BY (4)2
(5)4. QED
BY (5)1, (5)2, (5)3
(4)5. LL2ObservedAuthenticators' ⊆ MACType
(5)1. LL2ObservedAuthenticators ⊆ MACType
BY (2)1 DEF LL2TypeInvariant
(5)2. LL2ObservedAuthenticators' =
LL2ObservedAuthenticators ∪ {newAuthenticator}
BY (4)1 DEF newAuthenticator, newHistoryStateBinding, newHistorySummaryHash,
newStateHash, newHistorySummary, currentHistorySummary,
newPrivateStateEnc, sResult, privateState
(5)3. newAuthenticator ∈ MACType
BY (4)2
(5)4. QED
BY (5)1, (5)2, (5)3
(4)6. LL2Disk' ∈ LL2UntrustedStorageType
(5)1. LL2Disk ∈ LL2UntrustedStorageType
BY (2)1 DEF LL2TypeInvariant
(5)2. UNCHANGED LL2Disk
BY (4)1
(5)3. QED
BY (5)1, (5)2
(4)7. LL2RAM' ∈ LL2UntrustedStorageType
(5)1. LL2RAM' = {publicState → sResult.newPublicState,
privateStateEnc → newPrivateStateEnc,
historySummary → newHistorySummary,
authenticator → newAuthenticator}
BY (4)1 DEF newAuthenticator, newHistoryStateBinding, newHistorySummaryHash,
newStateHash, newHistorySummary, currentHistorySummary,
newPrivateStateEnc, sResult, privateState
(5)2. sResult.newPublicState ∈ PublicStateType

218
BY (4) 2
(5) 3. newPrivateStateEnc ∈ PrivateStateEncType
   BY (4) 2
(5) 4. newHistorySummary ∈ HistorySummaryType
   BY (4) 2 DEF newHistorySummary, currentHistorySummary
(5) 5. newAuthenticator ∈ MACType
   BY (4) 2
(5) 6. QED
   BY (5) 1, (5) 2, (5) 3, (5) 4, (5) 5 DEF LL2UntrustedStorageType
(4) 8. LL2NVRAM' ∈ LL2TrustedStorageType
(5) 1. LL2NVRAM' = [
   historySummaryAnchor → LL2NVRAM.historySummaryAnchor,
   symmetricKey → LL2NVRAM.symmetricKey,
   hashBarrier → LL2NVRAM.hashBarrier,
   extensionInProgress → TRUE]
   BY (4) 1 DEF LL2TypeInvariant, newHistorySummary
(5) 2. LL2NVRAM ∈ LL2TrustedStorageType
   BY (2) 1 DEF LL2TypeInvariant
(5) 3. TRUE ∈ BOOLEAN
   OBVIOUS
   BY (5) 1, (5) 2, (5) 3 DEF LL2TrustedStorageType
(4) 9. LL2SPCR' ∈ HashType
(5) 1. LL2SPCR' = newHistorySummary.extension
   BY (4) 1 DEF newHistorySummary,
   currentHistorySummary
(5) 2. newHistorySummary.extension ∈ HashType
   BY (4) 2
(5) 3. QED
   BY (5) 1, (5) 2
(4) 10. QED
   BY (4) 3, (4) 4, (4) 5, (4) 6, (4) 7, (4) 8, (4) 9 DEF LL2TypeInvariant
(3) 3. CASE LL2RepeatOperation
For a LL2RepeatOperation action, we just walk through the definitions. Type safety follows directly.
(4) 1. PICK input ∈ LL2AvailableInputs : LL2RepeatOperation!(input)!1
   BY (3) 3 DEF LL2RepeatOperation
(4) historySummaryHash ≜
   Hash(LL2RAM.historySummary.anchor, LL2RAM.historySummary.extension)
(4) stateHash ≜ Hash(LL2RAM.publicState, LL2RAM.privateStateEnc)
(4) historyStateBinding ≜ Hash(historySummaryHash, stateHash)
(4) newHistorySummary ≜ Successor(LL2RAM.historySummary, input, LL2NVRAM.hashBarrier)
(4) checkpointedHistorySummary ≜ Checkpoint(LL2RAM.historySummary)
(4) checkpointedNewCheckpointedHistorySummary ≜
   Successor(checkpointedHistorySummary, input, LL2NVRAM.hashBarrier)
(4) checkpointedNewHistorySummary ≜ Checkpoint(newHistorySummary)
(4) checkpointedNewCheckpointedHistorySummary ≜
   Checkpoint(newCheckpointedHistorySummary)
(4) privateState ≜ SymmetricDecrypt(LL2NVRAM.symmetricKey, LL2RAM.privateStateEnc)
(4) sResult ≜ Service(LL2RAM.publicState, privateState, input)
(4) newPrivateStateEnc ≜
   SymmetricEncrypt(LL2NVRAM.symmetricKey, sResult.newPrivateState)
(4) currentHistorySummary ≜ [
   anchor → LL2NVRAM.historySummaryAnchor,
extension $\mapsto$ LL2SPCR

(4) \(\text{currentHistorySummaryHash} \triangleq \text{Hash} (\text{LL2NVRAM.historySummaryAnchor}, \text{LL2SPCR})\)

(4) \(\text{newStateHash} \triangleq \text{Hash} (s\text{Result}.\text{newPublicState}, \text{newPrivateStateEnc})\)

(4) \(\text{newHistoryStateBinding} \triangleq \text{Hash} (\text{currentHistorySummaryHash}, \text{newStateHash})\)

(4) \(\text{newAuthenticator} \triangleq \text{GenerateMAC} (\text{LL2NVRAM.symmetricKey}, \text{newHistoryStateBinding})\)

(4.2) \(\land \ \text{historySummaryHash} \in \text{HashType}\)

\(\land \ \text{newStateHash} \in \text{HashType}\)

\(\land \ \text{newHistoryStateBinding} \in \text{HashType}\)

\(\land \ \text{newHistorySummary} \in \text{HistorySummaryType}\)

\(\land \ \text{newHistorySummary}.\text{anchor} \in \text{HashType}\)

\(\land \ \text{newHistorySummary}.\text{extension} \in \text{HashType}\)

\(\land \ \text{checkpointedHistorySummary} \in \text{HistorySummaryType}\)

\(\land \ \text{checkpointedHistorySummary}.\text{anchor} \in \text{HashType}\)

\(\land \ \text{checkpointedHistorySummary}.\text{extension} \in \text{HashType}\)

\(\land \ \text{newCheckpointedHistorySummary} \in \text{HistorySummaryType}\)

\(\land \ \text{newCheckpointedHistorySummary}.\text{anchor} \in \text{HashType}\)

\(\land \ \text{newCheckpointedHistorySummary}.\text{extension} \in \text{HashType}\)

\(\land \ \text{newCheckpointedHistorySummary}.\text{historyStateBinding}.\text{hash} \in \text{HashType}\)

\(\land \ \text{newCheckpointedHistorySummary}.\text{historyStateBinding}.\text{extension} \in \text{HashType}\)

\(\land \ \text{newCheckpointedHistorySummary}.\text{historyStateBinding} \in \text{HashType}\)

\(\land \ \text{newCheckpointedHistorySummary}.\text{historyStateBinding}.\text{hash} \in \text{HashType}\)

\(\land \ \text{newCheckpointedHistorySummary}.\text{historyStateBinding}.\text{extension} \in \text{HashType}\)

\(\land \ \text{privateState} \in \text{PrivateStateType}\)

\(\land \ s\text{Result} \in \text{ServiceResultType}\)

\(\land \ s\text{Result}.\text{newPublicState} \in \text{PublicStateType}\)

\(\land \ s\text{Result}.\text{newPrivateState} \in \text{PrivateStateType}\)

\(\land \ s\text{Result}.\text{output} \in \text{OutputType}\)

\(\land \ \text{newPrivateStateEnc} \in \text{PrivateStateEncType}\)

\(\land \ \text{currentHistorySummary} \in \text{HistorySummaryType}\)

\(\land \ \text{currentHistorySummary}.\text{anchor} \in \text{HashType}\)

\(\land \ \text{currentHistorySummary}.\text{extension} \in \text{HashType}\)

\(\land \ \text{currentHistorySummary}.\text{hash} \in \text{HashType}\)

\(\land \ \text{newStateHash} \in \text{HashType}\)

\(\land \ \text{newHistoryStateBinding} \in \text{HashType}\)

\(\land \ \text{newAuthenticator} \in \text{MACType}\)

(5.1) \(\text{input} \in \text{LL2AvailableInputs}\)

\(\text{by} \ (4.1)\)

(5.2) \(\text{LL2TypeInvariant}\)

\(\text{by} \ (2.1)\)

(5.3) \(\text{QED}\)

\(\text{by} \ (5.1), (5.2), \text{LL2RepeatOperationDefsTypeSafeLemma}\)

(4) \(\text{hide def} \ \text{historySummaryHash}, \text{stateHash}, \text{historyStateBinding}, \text{newHistorySummary}, \text{checkpointedHistorySummary}, \text{checkpointedNewHistorySummary}, \text{checkpointedNewCheckpointedHistorySummary}, \text{privateState}, s\text{Result}, \text{newPrivateStateEnc}, \text{currentHistorySummary}, \text{currentHistorySummary}.\text{hash}, \text{newStateHash}, \text{newHistoryStateBinding}, \text{newAuthenticator}\)

(4.3) \(\text{LL2AvailableInputs}' \subseteq \text{InputType}\)

(5.1) \(\text{LL2AvailableInputs} \subseteq \text{InputType}\)

\(\text{by} \ (2.1) \text{def LL2TypeInvariant}\)

(5.2) \(\text{UNCHANGED LL2AvailableInputs}\)
\[
\text{LL}_2 \text{ObservedOutputs}' \subseteq \text{OutputType} \\
\text{LL}_2 \text{ObservedAuthenticators}' \subseteq \text{MACType} \\
\text{LL}_2 \text{Disk}' \in \text{LL}_2 \text{UntrustedStorageType} \\
\text{LL}_2 \text{RAM}' \in \text{LL}_2 \text{UntrustedStorageType} \\
\text{LL}_2 \text{NVRAM}' \in \text{LL}_2 \text{TrustedStorageType}
\]
For a LL2TakeCheckpoint action, we just walk through the definitions. Type safety follows directly.

(4) newHistorySummaryAnchor \(\triangleq\) Hash(LL2NVRAM.historySummaryAnchor, LL2SPCR)
(4.1) LL2TypeInvariant ∈ HashType
(5.1) UNCHANGED LL2SPCR
(5.2) QED
(5.3) BY (4.1) LL2TakeCheckpointDefSSTypeSafeLemma
(4.2) LL2AvailableInputs' ⊆ InputType
(5.1) LL2AvailableInputs ⊆ InputType
(5.2) UNCHANGED LL2AvailableInputs
(5.3) QED
(5.4) BY (4.1) LL2TakeCheckpoint
(4.3) LL2ObservedOutputs' ⊆ OutputType
(5.1) LL2ObservedOutputs ⊆ OutputType
(5.2) UNCHANGED LL2ObservedOutputs
(5.3) QED
(5.4) BY (4.1) LL2TakeCheckpoint
(4.4) LL2ObservedAuthenticators' ⊆ MACType
(5.1) LL2ObservedAuthenticators ⊆ MACType
(5.2) UNCHANGED LL2ObservedAuthenticators
(5.3) QED
(5.4) BY (4.1) LL2TakeCheckpoint
(4.5) LL2Disk' ∈ LL2UntrustedStorageType
(5.1) LL2Disk ∈ LL2UntrustedStorageType
(5.2) UNCHANGED LL2Disk
(5.3) QED
(5.4) BY (4.1) LL2TakeCheckpoint
(4.6) LL2RAM' ∈ LL2UntrustedStorageType
(5.1) LL2RAM ∈ LL2UntrustedStorageType
(5.2) BY (2.1) LL2TypeInvariant
(5) 2. UNCHANGED $LL2RAM$
    BY (3) 4 DEFN $LL2TakeCheckpoint$
(5) 3. QED
    BY (5) 1, (5) 2
(4) 7. $LL2NVRAM' \in LL2TrustedStorageType$
(5) 1. $LL2NVRAM \in LL2TrustedStorageType$
    BY (2) 1 DEFN $LL2TypeInvariant$
(5) 2. $LL2NVRAM' = [
        historySummaryAnchor \mapsto newHistorySummaryAnchor,
        symmetricKey \mapsto LL2NVRAM.symmetricKey,
        hashBarrier \mapsto LL2NVRAM.hashBarrier,
        extensionInProgress \mapsto false
    ]$
    BY (3) 4 DEFN $LL2TakeCheckpoint$, newHistorySummaryAnchor
(5) 3. newHistorySummaryAnchor \in HashType
    BY (4) 1
(5) 4. FALSE \in BOOLEAN
    OBVIOUS
(5) 5. QED
    BY (5) 1, (5) 2, (5) 3, (5) 4 DEFN $LL2TrustedStorageType$
(4) 8. $LL2SPCR' \in HashType$
(5) 1. $LL2SPCR \in HashType$
    BY (2) 1 DEFN $LL2TypeInvariant$
(5) 2. UNCHANGED $LL2SPCR$
    BY (3) 4 DEFN $LL2TakeCheckpoint$
(5) 3. QED
    BY (5) 1, (5) 2
(4) 9. QED
    BY (4) 2, (4) 3, (4) 4, (4) 5, (4) 6, (4) 7, (4) 8 DEFN $LL2TypeInvariant$
(3) 5. CASE $LL2Restart$
    For a $LL2Restart$ action, we just walk through the definitions. Type safety follows directly.
(4) 1. PICK $untrustedStorage \in LL2UntrustedStorageType,$
    randomSymmetricKey \in SymmetricKeyType \setminus \{LL2NVRAM.symmetricKey\},
    hash \in HashType : $LL2Restart!(untrustedStorage, randomSymmetricKey, hash)$
    BY (3) 5 DEFN $LL2Restart$
(4) 2. $LL2AvailableInputs' \subseteq InputType$
    (5) 1. $LL2AvailableInputs \subseteq InputType$
        BY (2) 1 DEFN $LL2TypeInvariant$
    (5) 2. UNCHANGED $LL2AvailableInputs$
        BY (4) 1
    (5) 3. QED
        BY (5) 1, (5) 2
(4) 3. $LL2ObservedOutputs' \subseteq OutputType$
    (5) 1. $LL2ObservedOutputs \subseteq OutputType$
        BY (2) 1 DEFN $LL2TypeInvariant$
    (5) 2. UNCHANGED $LL2ObservedOutputs$
        BY (4) 1
    (5) 3. QED
        BY (5) 1, (5) 2
(4) 4. $LL2ObservedAuthenticators' \subseteq MACType$
    (5) 1. $LL2ObservedAuthenticators \subseteq MACType$
        BY (2) 1 DEFN $LL2TypeInvariant$
(5.2) UNCHANGED $LL2\text{ObservedAuthenticators}$
   BY (4)1
(5.3) QED
   BY (5)1, (5)2
(4.5) $LL2\text{Disk}' \in LL2\text{UntrustedStorageType}$
   (5.1) $LL2\text{Disk} \in LL2\text{UntrustedStorageType}$
      BY (2)1 DEF $LL2\text{TypeInvariant}$
   (5.2) UNCHANGED $LL2\text{Disk}$
      BY (4)1
(5.3) QED
      BY (5)1, (5)2
(4.6) $LL2\text{RAM}' \in LL2\text{UntrustedStorageType}$
   (5.1) $LL2\text{Disk} \in LL2\text{UntrustedStorageType}$
      BY (2)1 DEF $LL2\text{TypeInvariant}$
   (5.2) $LL2\text{RAM}' = \text{untrustedStorage}$
      BY (4)1
   (5.3) $\text{untrustedStorage} \in LL2\text{UntrustedStorageType}$
      BY (4)1
   (5.4) QED
      BY (5)1, (5)2, (5)3
(4.7) $LL2\text{NVRAM}' \in LL2\text{TrustedStorageType}$
   (5.1) $LL2\text{NVRAM} \in LL2\text{TrustedStorageType}$
      BY (2)1 DEF $LL2\text{TypeInvariant}$
   (5.2) UNCHANGED $LL2\text{NVRAM}$
      BY (4)1
   (5.3) QED
      BY (5)1, (5)2
(4.8) $LL2\text{SPCR}' \in \text{HashType}$
   (5.1) $LL2\text{SPCR}' = \text{BaseHashValue}$
      BY (4)1
   (5.2) $\text{BaseHashValue} \in \text{HashType}$
      BY $\text{BaseHashValueTypeSafe}$
   (5.3) QED
      BY (5)1, (5)2
(4.9) QED
      BY (4)2, (4)3, (4)4, (4)5, (4)6, (4)7, (4)8 DEF $LL2\text{TypeInvariant}$
(3)6. CASE $LL2\text{ReadDisk}$
   Type safety is straightforward for a $LL2\text{ReadDisk}$ action.

(4.1) $LL2\text{AvailableInputs}' \subseteq \text{InputType}$
   (5.1) $LL2\text{AvailableInputs} \subseteq \text{InputType}$
      BY (2)1 DEF $LL2\text{TypeInvariant}$
   (5.2) UNCHANGED $LL2\text{AvailableInputs}$
      BY (3)6 DEF $LL2\text{ReadDisk}$
   (5.3) QED
      BY (5)1, (5)2
(4.2) $LL2\text{ObservedOutputs}' \subseteq \text{OutputType}$
   (5.1) $LL2\text{ObservedOutputs} \subseteq \text{OutputType}$
      BY (2)1 DEF $LL2\text{TypeInvariant}$
   (5.2) UNCHANGED $LL2\text{ObservedOutputs}$
      BY (3)6 DEF $LL2\text{ReadDisk}$
   (5.3) QED
      BY (5)1, (5)2
(4.3) \textit{LL2ObservedAuthenticators}' \subseteq \textit{MACType}

(5.1) \textit{LL2ObservedAuthenticators} \subseteq \textit{MACType}
    \textit{LL2TypeInvariant}

(5.2) \textit{LL2ObservedAuthenticators}
    \textit{LL2ReadDisk}

(5.3) \textit{QED}
    \textit{LL2TypeInvariant}

(4.4) \textit{LL2Disk'} \in \textit{LL2UntrustedStorageType}

(5.1) \textit{LL2Disk} \in \textit{LL2UntrustedStorageType}
    \textit{LL2TypeInvariant}

(5.2) \textit{LL2Disk} \equiv \textit{LL2Disk}
    \textit{LL2ReadDisk}

(5.3) \textit{QED}
    \textit{LL2TypeInvariant}

(4.5) \textit{LL2RAM'} \in \textit{LL2UntrustedStorageType}

(5.1) \textit{LL2Disk} \in \textit{LL2UntrustedStorageType}
    \textit{LL2TypeInvariant}

(5.2) \textit{LL2RAM} = \textit{LL2Disk}
    \textit{LL2ReadDisk}

(5.3) \textit{QED}
    \textit{LL2TypeInvariant}

(4.6) \textit{LL2NVRAM'} \in \textit{LL2TrustedStorageType}

(5.1) \textit{LL2NVRAM} \in \textit{LL2TrustedStorageType}
    \textit{LL2TypeInvariant}

(5.2) \textit{LL2NVRAM} = \textit{LL2Disk}
    \textit{LL2ReadDisk}

(5.3) \textit{QED}
    \textit{LL2TypeInvariant}

(4.7) \textit{LL2SPCR'} \in \textit{HashType}

(5.1) \textit{LL2SPCR} \in \textit{HashType}
    \textit{LL2TypeInvariant}

(5.2) \textit{LL2SPCR} \equiv \textit{LL2Disk}
    \textit{LL2ReadDisk}

(5.3) \textit{QED}
    \textit{LL2TypeInvariant}

(4.8) \textit{QED}
    \textit{LL2WriteDisk}

(3.7) \textit{QED}

Type safety is straightforward for a \textit{LL2WriteDisk} action.

(4.1) \textit{LL2AvailableInputs}' \subseteq \textit{InputType}

(5.1) \textit{LL2AvailableInputs} \subseteq \textit{InputType}
    \textit{LL2TypeInvariant}

(5.2) \textit{LL2AvailableInputs}
    \textit{LL2WriteDisk}

(5.3) \textit{QED}
    \textit{LL2TypeInvariant}

(4.2) \textit{LL2ObservedOutputs}' \subseteq \textit{OutputType}

(5.1) \textit{LL2ObservedOutputs} \subseteq \textit{OutputType}
    \textit{LL2TypeInvariant}

(5.2) \textit{LL2ObservedOutputs}
    \textit{LL2WriteDisk}

(5.3) \textit{QED}
(4.3) \( \text{LL2ObservedAuthenticators}' \subseteq \text{MACType} \\
(5.1) \text{LL2ObservedAuthenticators} \subseteq \text{MACType} \\
\text{by } (2.1) \text{ DEF } \text{LL2TypeInvariant} \\
(5.2) \text{unchanged } \text{LL2ObservedAuthenticators} \\
\text{by } (3.7) \text{ DEF } \text{LL2WriteDisk} \\
(5.3) \text{QED} \\
\text{by } (5.1), (5.2) \\
(4.4) \text{LL2Disk}' \in \text{LL2UntrustedStorageType} \\
(5.1) \text{LL2RAM} \in \text{LL2UntrustedStorageType} \\
\text{by } (2.1) \text{ DEF } \text{LL2TypeInvariant} \\
(5.2) \text{unchanged } \text{LL2RAM} \\
\text{by } (3.7) \text{ DEF } \text{LL2WriteDisk} \\
(5.3) \text{QED} \\
\text{by } (5.1), (5.2) \\
(4.5) \text{LL2RAM}' \in \text{LL2UntrustedStorageType} \\
(5.1) \text{LL2RAM} \in \text{LL2UntrustedStorageType} \\
\text{by } (2.1) \text{ DEF } \text{LL2TypeInvariant} \\
(5.2) \text{unchanged } \text{LL2RAM} \\
\text{by } (3.7) \text{ DEF } \text{LL2WriteDisk} \\
(5.3) \text{QED} \\
\text{by } (5.1), (5.2) \\
(4.6) \text{LL2NVRAM}' \in \text{LL2TrustedStorageType} \\
(5.1) \text{LL2NVRAM} \in \text{LL2TrustedStorageType} \\
\text{by } (2.1) \text{ DEF } \text{LL2TypeInvariant} \\
(5.2) \text{unchanged } \text{LL2NVRAM} \\
\text{by } (3.7) \text{ DEF } \text{LL2WriteDisk} \\
(5.3) \text{QED} \\
\text{by } (5.1), (5.2) \\
(4.7) \text{LL2SPCR}' \in \text{HashType} \\
(5.1) \text{LL2SPCR} \in \text{HashType} \\
\text{by } (2.1) \text{ DEF } \text{LL2TypeInvariant} \\
(5.2) \text{unchanged } \text{LL2SPCR} \\
\text{by } (3.7) \text{ DEF } \text{LL2WriteDisk} \\
(5.3) \text{QED} \\
\text{by } (5.1), (5.2) \\
(4.8) \text{QED} \\
\text{by } (4.1), (4.2), (4.3), (4.4), (4.5), (4.6), (4.7) \text{ DEF } \text{LL2TypeInvariant} \\
(3.8) \text{case } \text{LL2CorruptRAM} \\
\text{Type safety is straightforward for a } \text{LL2CorruptRAM} \text{ action.} \\
(4.1) \text{pick } \text{untrustedStorage} \in \text{LL2UntrustedStorageType}, \\
\text{fakeSymmetricKey} \in \text{SymmetricKeyType} \setminus \{\text{LL2NVRAM}\text{.symmetricKey}\}, \\
\text{hash} \in \text{HashType} : \\
\text{LL2CorruptRAM}(\text{untrustedStorage}, \text{fakeSymmetricKey}, \text{hash}) \\
\text{by } (3.8) \text{ DEF } \text{LL2CorruptRAM} \\
(4.2) \text{LL2AvailableInputs}' \subseteq \text{InputType} \\
(5.1) \text{LL2AvailableInputs} \subseteq \text{InputType} \\
\text{by } (2.1) \text{ DEF } \text{LL2TypeInvariant} \\
(5.2) \text{unchanged } \text{LL2AvailableInputs} \\
\text{by } (4.1) \\
(5.3) \text{QED} \\
\text{by } (5.1), (5.2)
(4.3) \( LL2ObservedOutputs' \subseteq OutputType \)
(5.1) \( LL2ObservedOutputs \subseteq OutputType \)
BY \( \langle 2 \rangle 1 \) DEF \( LL2TypeInvariant \)
(5.2) UNCHANGED \( LL2ObservedOutputs \)
BY \( \langle 4 \rangle 1 \)
(5.3) QED
BY \( \langle 5 \rangle 1, \langle 5 \rangle 2 \)

(4.4) \( LL2ObservedAuthenticators' \subseteq MACType \)
(5.1) \( LL2ObservedAuthenticators \subseteq MACType \)
BY \( \langle 2 \rangle 1 \) DEF \( LL2TypeInvariant \)
(5.2) UNCHANGED \( LL2ObservedAuthenticators \)
BY \( \langle 4 \rangle 1 \)
(5.3) QED
BY \( \langle 5 \rangle 1, \langle 5 \rangle 2 \)

(4.5) \( LL2Disk' \in LL2UntrustedStorageType \)
(5.1) \( LL2Disk \in LL2UntrustedStorageType \)
BY \( \langle 2 \rangle 1 \) DEF \( LL2TypeInvariant \)
(5.2) UNCHANGED \( LL2Disk \)
BY \( \langle 4 \rangle 1 \)
(5.3) QED
BY \( \langle 5 \rangle 1, \langle 5 \rangle 2 \)

(4.6) \( LL2RAM' \in LL2UntrustedStorageType \)
(5.1) \( untrustedStorage \in LL2UntrustedStorageType \)
BY \( \langle 4 \rangle 1 \)
(5.2) \( LL2RAM' = untrustedStorage \)
BY \( \langle 4 \rangle 1 \)
(5.3) QED
BY \( \langle 5 \rangle 1, \langle 5 \rangle 2 \)

(4.7) \( LL2NVRAM' \in LL2TrustedStorageType \)
(5.1) \( LL2NVRAM \in LL2TrustedStorageType \)
BY \( \langle 2 \rangle 1 \) DEF \( LL2TypeInvariant \)
(5.2) UNCHANGED \( LL2NVRAM \)
BY \( \langle 4 \rangle 1 \)
(5.3) QED
BY \( \langle 5 \rangle 1, \langle 5 \rangle 2 \)

(4.8) \( LL2SPCR' \in HashType \)
(5.1) \( LL2SPCR \in HashType \)
BY \( \langle 2 \rangle 1 \) DEF \( LL2TypeInvariant \)
(5.2) UNCHANGED \( LL2SPCR \)
BY \( \langle 4 \rangle 1 \)
(5.3) QED
BY \( \langle 5 \rangle 1, \langle 5 \rangle 2 \)

(4.9) QED
BY \( \langle 4 \rangle 2, \langle 4 \rangle 3, \langle 4 \rangle 4, \langle 4 \rangle 5, \langle 4 \rangle 6, \langle 4 \rangle 7, \langle 4 \rangle 8 \) DEF \( LL2TypeInvariant \)

For a \( LL2CorruptSPCR \) action, we just walk through the definitions. Type safety follows directly.

(4.1) PICK \( fakeHash \in HashDomain : LL2CorruptSPCR!(fakeHash)!1 \)
BY \( \langle 3 \rangle 9 \) DEF \( LL2CorruptSPCR \)
(4) \( newHistorySummaryExtension \triangleq Hash(LL2SPCR, fakeHash) \)
(4.2) \( newHistorySummaryExtension \in HashType \)
(5.1) \( LL2TypeInvariant \)
BY \( \langle 2 \rangle 1 \)
(5.2) QED
BY (5.1), LL2CorruptSPCRCorruptSPCRDefsTypeSafeLemma

(4.4) HIDE DEF newHistorySummaryExtension

(4.3) LL2AvailableInputs' ⊆ InputType
(5.1) LL2AvailableInputs ⊆ InputType
BY (2.1) DEF LL2TypeInvariant
(5.2) UNCHANGED LL2AvailableInputs
BY (4.1)
(5.3) QED
BY (5.1), (5.2)

(4.4) LL2ObservedOutputs' ⊆ OutputType
(5.1) LL2ObservedOutputs ⊆ OutputType
BY (2.1) DEF LL2TypeInvariant
(5.2) UNCHANGED LL2ObservedOutputs
BY (4.1)
(5.3) QED
BY (5.1), (5.2)

(4.5) LL2ObservedAuthenticators' ⊆ MACType
(5.1) LL2ObservedAuthenticators ⊆ MACType
BY (2.1) DEF LL2TypeInvariant
(5.2) UNCHANGED LL2ObservedAuthenticators
BY (4.1)
(5.3) QED
BY (5.1), (5.2)

(4.6) LL2Disk' ∈ LL2UntrustedStorageType
(5.1) LL2Disk ∈ LL2UntrustedStorageType
BY (2.1) DEF LL2TypeInvariant
(5.2) UNCHANGED LL2Disk
BY (4.1)
(5.3) QED
BY (5.1), (5.2)

(4.7) LL2RAM' ∈ LL2UntrustedStorageType
(5.1) LL2RAM ∈ LL2UntrustedStorageType
BY (2.1) DEF LL2TypeInvariant
(5.2) UNCHANGED LL2RAM
BY (4.1)
(5.3) QED
BY (5.1), (5.2)

(4.8) LL2NVRAM' ∈ LL2TrustedStorageType
(5.1) LL2NVRAM ∈ LL2TrustedStorageType
BY (2.1) DEF LL2TypeInvariant
(5.2) UNCHANGED LL2NVRAM
BY (4.1)
(5.3) QED
BY (5.1), (5.2)

(4.9) LL2SPCR' ∈ HashType
(5.1) newHistorySummaryExtension ∈ HashType
BY (4.2)
(5.2) LL2SPCR' = newHistorySummaryExtension
BY (4.1) DEF newHistorySummaryExtension
(5.3) QED
BY (5.1), (5.2)
Using the Inv1 proof rule, the base case and the induction step together imply that the invariant always holds.

(2)1. $LL2TypeInvariant \land \Box [LL2Next]_{LL2Vars} \Rightarrow \Box LL2TypeInvariant$
   BY (1)2, Inv1

(2)2. QED
   BY (2)1, (1)1 DEF LL2Spec
This module states and proves several lemmas about the operators defined in the `LL2Refinement` module.

This module includes the following theorems:
- `HistorySummaryRecordCompositionLemma`
- `LL1DiskRecordCompositionLemma`
- `LL1RAMRecordCompositionLemma`
- `LL1NVRAMRecordCompositionLemma`
- `CheckpointHasBaseExtensionLemma`
- `SuccessorHasNonBaseExtensionLemma`
- `HistorySummariesMatchUniqueLemma`
- `AuthenticatorsMatchUniqueLemma`
- `AuthenticatorSetsMatchUniqueLemma`
- `LL2NVRAMLogicalHistorySummaryTypeSafe`
- `AuthenticatorInSetLemma`
- `AuthenticatorGeneratedLemma`
- `AuthenticatorValidatedLemma`
- `HistorySummariesMatchAcrossCheckpointLemma`
publicState \rightarrow LL1Disk\_publicState',
privateStateEnc \rightarrow LL1Disk\_privateStateEnc',
historySummary \rightarrow LL1Disk\_historySummary',
authenticator \rightarrow LL1Disk\_authenticator'

(1)1. LL1Disk \in LL1\_UntrustedStorageType \Rightarrow
\begin{align*}
LL1Disk &= [ \\
publicState &\rightarrow LL1Disk\_publicState, \\
privateStateEnc &\rightarrow LL1Disk\_privateStateEnc, \\
historySummary &\rightarrow LL1Disk\_historySummary, \\
authenticator &\rightarrow LL1Disk\_authenticator]
\end{align*}

(2)1. HAVE LL1Disk \in LL1\_UntrustedStorageType

(2) \text{ll1disk} \triangleq [ \\
publicState &\rightarrow LL1Disk\_publicState, \\
privateStateEnc &\rightarrow LL1Disk\_privateStateEnc, \\
historySummary &\rightarrow LL1Disk\_historySummary, \\
authenticator &\rightarrow LL1Disk\_authenticator]

(2)2. ll1disk = 
\quad [i \in \{\text{"publicState"}, \\
\text{"privateStateEnc"}, \\
\text{"historySummary"}, \\
\text{"authenticator"} \} \\
\Rightarrow ll1disk[i]]

\text{OBVIOUS}

(2)3. LL1Disk = 
\quad [i \in \{\text{"publicState"}, \\
\text{"privateStateEnc"}, \\
\text{"historySummary"}, \\
\text{"authenticator"} \} \\
\Rightarrow LL1Disk[i]]

BY (2)1 DEF LL1\_UntrustedStorageType

(2)4. QED

BY (2)2, (2)3

(1)2. LL1Disk' \in LL1\_UntrustedStorageType \Rightarrow
\begin{align*}
LL1Disk' &= [ \\
publicState &\rightarrow LL1Disk\_publicState', \\
privateStateEnc &\rightarrow LL1Disk\_privateStateEnc', \\
historySummary &\rightarrow LL1Disk\_historySummary', \\
authenticator &\rightarrow LL1Disk\_authenticator'
\end{align*}

(2)1. HAVE LL1Disk' \in LL1\_UntrustedStorageType

(2) ll1diskprime \triangleq [ \\
publicState &\rightarrow LL1Disk\_publicState', \\
privateStateEnc &\rightarrow LL1Disk\_privateStateEnc', \\
historySummary &\rightarrow LL1Disk\_historySummary', \\
authenticator &\rightarrow LL1Disk\_authenticator'
\end{align*}

(2)2. ll1diskprime = 
\quad [i \in \{\text{"publicState"}, \\
\text{"privateStateEnc"}, \\
\text{"historySummary"}, \\
\text{"authenticator"} \} \\
\Rightarrow ll1diskprime[i]]

\text{OBVIOUS}

(2)3. LL1Disk' =
The \textit{LL1RAMRecordCompositionLemma} is a formality needed for the prover to compose an \textit{LL1RAM} record from fields of the appropriate type.

\textbf{THEOREM} \(\text{LL1RAMRecordCompositionLemma} \triangleq \)

\[
\wedge \text{LL1RAM} \in \text{LL1UntrustedStorageType} \Rightarrow \\
\text{LL1RAM} = [
\text{publicState} \mapsto \text{LL1RAM.publicState}, \\
\text{privateStateEnc} \mapsto \text{LL1RAM.privateStateEnc}, \\
\text{historySummary} \mapsto \text{LL1RAM.historySummary}, \\
\text{authenticator} \mapsto \text{LL1RAM.authenticator}]
\]

\[
\wedge \text{LL1RAM'} \in \text{LL1UntrustedStorageType} \Rightarrow \\
\text{LL1RAM'} = [
\text{publicState} \mapsto \text{LL1RAM.publicState'}, \\
\text{privateStateEnc} \mapsto \text{LL1RAM.privateStateEnc'}, \\
\text{historySummary} \mapsto \text{LL1RAM.historySummary'}, \\
\text{authenticator} \mapsto \text{LL1RAM.authenticator'}]
\]

\(\langle 1 \rangle 1. \text{LL1RAM} \in \text{LL1UntrustedStorageType} \Rightarrow \\
\text{LL1RAM} = [
\text{publicState} \mapsto \text{LL1RAM.publicState}, \\
\text{privateStateEnc} \mapsto \text{LL1RAM.privateStateEnc}, \\
\text{historySummary} \mapsto \text{LL1RAM.historySummary}, \\
\text{authenticator} \mapsto \text{LL1RAM.authenticator}]
\]

\(\langle 2 \rangle 1. \text{HAVE } \text{LL1RAM} \in \text{LL1UntrustedStorageType}
\)

\(\langle 2 \rangle \text{ll1ram} \triangleq [\]
\[\text{publicState} \mapsto \text{LL1RAM.publicState}, \\
\text{privateStateEnc} \mapsto \text{LL1RAM.privateStateEnc}, \\
\text{historySummary} \mapsto \text{LL1RAM.historySummary}, \\
\text{authenticator} \mapsto \text{LL1RAM.authenticator}]
\]

\(\langle 2 \rangle 2. \text{ll1ram} = \\
\[i \in \{ \text{"publicState"}, \\
\text{"privateStateEnc"}, \\
\text{"historySummary"}, \\
\text{"authenticator"} \} \\
\mapsto \text{ll1ram}[i]\]
\)

\textbf{OBVIOUS}

\(\langle 2 \rangle 3. \text{LL1RAM} = \\
\[i \in \{ \text{"publicState"}, \\
\text{"privateStateEnc"}, \\
\text{"historySummary"}, \\
\text{"authenticator"} \} \\
\mapsto \text{LL1RAM[i]}\]
\)
The \textit{LL1NVRAMRecordCompositionLemma} is a formality needed for the prover to compose an \textit{LL1NVRAM} record from fields of the appropriate type.

\begin{theorem}
\textbf{LL1NVRAMRecordCompositionLemma} \triangleq \\
\text{\textit{LL1NVRAM}} \in \text{\textit{LL1TrustedStorageType}} \Rightarrow \\
\text{\textit{LL1NVRAM}} = [ \\
\text{\textit{historySummary}} \mapsto \text{\textit{LL1NVRAM}.historySummary}, \\
\text{\textit{symmetricKey}} \mapsto \text{\textit{LL1NVRAM}.symmetricKey} ] \\
\wedge \\
\text{\textit{LL1NVRAM'}} \in \text{\textit{LL1TrustedStorageType}} \Rightarrow \\
\text{\textit{LL1NVRAM'}} = [ \\
\text{\textit{historySummary}} \mapsto \text{\textit{LL1NVRAM}.historySummary'}, \\
\text{\textit{symmetricKey}} \mapsto \text{\textit{LL1NVRAM}.symmetricKey'} ]
\end{theorem}

\begin{enumerate}
\item\textit{1.} \textit{LL1NVRAM} \in \textit{LL1TrustedStorageType} \Rightarrow \\
\textit{LL1NVRAM} = [ \\
\textit{historySummary} \mapsto \textit{LL1NVRAM}.\textit{historySummary}, \\
\textit{symmetricKey} \mapsto \textit{LL1NVRAM}.\textit{symmetricKey} ]
\item\textit{2.} \textit{LL1NVRAM} \in \textit{LL1TrustedStorageType} \Rightarrow \\
\textit{LL1NVRAM} = [ \\
\textit{historySummary} \mapsto \textit{LL1NVRAM}.\textit{historySummary}, \\
\textit{symmetricKey} \mapsto \textit{LL1NVRAM}.\textit{symmetricKey} ]
\end{enumerate}
The CheckpointHasBaseExtensionLemma proves that a history summary produced by the Checkpoint function has an extension field that equals the base hash value.

THEOREM CheckpointHasBaseExtensionLemma \( \Delta \)
\[ \forall \text{historySummary} \in \text{HistorySummaryType} : \text{Checkpoint}(\text{historySummary}).\text{extension} = \text{BaseHashValue} \]

\langle 1 \rangle. \text{TAKE} \text{historySummary} \in \text{HistorySummaryType} \\
\langle 1 \rangle. \text{checkpointedAnchor} \( \Delta \) \text{Hash}(\text{historySummary}.\text{anchor}, \text{historySummary}.\text{extension}) \\
\langle 1 \rangle. \text{checkpointedHistorySummary} \( \Delta \) [ \\
\langle 1 \rangle. \text{anchor} ↦ \text{checkpointedAnchor}, \\
\langle 1 \rangle. \text{extension} ↦ \text{BaseHashValue} ] \\
\langle 1 \rangle. \text{HIDE DEF} \text{checkpointedAnchor}, \text{checkpointedHistorySummary} \\
\langle 1 \rangle. \text{CASE} \text{historySummary}.\text{extension} \neq \text{BaseHashValue} \\
\langle 2 \rangle. \text{Checkpoitnt}(\text{historySummary}) = \text{historySummary} \\
\langle 2 \rangle. \text{QED} \\
\langle 2 \rangle. \text{CASE} \text{historySummary}.\text{extension} = \text{BaseHashValue} \\
\langle 2 \rangle. \text{Checkpoitnt}(\text{historySummary}) = \text{checkpointedHistorySummary} \\
\langle 2 \rangle. \text{QED} \\
\langle 1 \rangle. \text{QED} \\
\langle 1 \rangle. \text{QED} \\
\langle 1 \rangle. \text{QED} \\
\langle 2 \rangle. \text{QED} \\
\langle 1 \rangle. \text{QED} \\
\langle 2 \rangle. \text{QED}
The `SuccessorHasNonBaseExtensionLemma` proves that a history summary produced by the `Successor` function has an extension field that does not equal the base hash value.

**THEOREM SuccessorHasNonBaseExtensionLemma** \( \triangleq \)
\[
\forall \text{historySummary} \in \text{HistorySummaryType}, \text{input} \in \text{InputType}, \text{hashBarrier} \in \text{HashType} : \\
\text{Successor} (\text{historySummary}, \text{input}, \text{hashBarrier}).\text{extension} \neq \text{BaseHashValue}
\]

\((1)\). **TAKE** `historySummary` \( \in \text{HistorySummaryType}, \text{input} \in \text{InputType}, \text{hashBarrier} \in \text{HashType}
\]
\[(1)\] `securedInput` \( \triangleq \) `Hash(hashBarrier, input)`
\[(1)\] `newAnchor` \( \triangleq \) `historySummary.\text{anchor}`
\[(1)\] `newExtension` \( \triangleq \) `Hash(historySummary.\text{extension}, \text{securedInput})`
\[(1)\] `newHistorySummary` \( \triangleq \) `[anchor \mapsto \text{newAnchor}, \text{extension} \mapsto \text{newExtension}]`
\[(1)\] **HIDE DEF** `securedInput`, `newAnchor`, `newExtension`, `newHistorySummary`
\[(1)\]. **newExtension** \( \neq \) `BaseHashValue`
\[(2)\]. **newExtension** \( \in \) `HashDomain`
\[(3)\]. `historySummary.\text{extension} \in \text{HashType}
\[(4)\]. `historySummary` \( \in \) `HistorySummaryType`
\[(4)\]. **BY** `(1)\)
\[(4)\]. **QED**
\[(3)\]. **QED**
\[(2)\]. **securedInput** \( \in \) `HashDomain`
\[(3)\]. `securedInput` \( \in \) `HashType`
\[(5)\]. `hashBarrier` \( \in \) `HashDomain`
\[(5)\]. **BY** `(1)\)
\[(5)\]. **QED**
\[(4)\]. `input` \( \in \) `HashDomain`
\[(5)\]. `input` \( \in \) `InputType`
\[(5)\]. **BY** `(1)\)
\[(5)\]. **QED**
\[(4)\]. **QED**
\[(3)\]. **QED**
\[(2)\]. **BY** `(2)\), `(2)\)
\[(1)\]. **BaseHashValueUnique**
\[(1)\]. **newExtension** \( = \) `Successor(historySummary, input, hashBarrier).\text{extension}`
\[(1)\]. **BY** `Successor`, `newExtension`, `securedInput`
\[(1)\]. **QED**
\[(1)\]. **BY** `(1)\), `(1)\)

235
The `HistorySummariesMatchUniqueLemma` asserts that there is a unique Memoir-Basic history summary that matches a particular Memoir-Opt history summary.

**THEOREM HistorySummariesMatchUniqueLemma** \(\triangleq\)
\[\forall \ll1\text{HistorySummary}1, \ll1\text{HistorySummary}2 \in \text{HashType},\]
\[\ll2\text{HistorySummary} \in \text{HistorySummaryType},\]
\[\text{hashBarrier} \in \text{HashType} :\]
\[(\wedge \text{HistorySummariesMatch}(\ll1\text{HistorySummary}1, \ll2\text{HistorySummary}, \text{hashBarrier})\]
\[\wedge \text{HistorySummariesMatch}(\ll1\text{HistorySummary}2, \ll2\text{HistorySummary}, \text{hashBarrier}))\]
\[\Rightarrow \ll1\text{HistorySummary}1 = \ll1\text{HistorySummary}2\]

It is convenient to define a predicate that captures the quantified expression of the theorem.

\[\langle 1 \rangle\text{ HistorySummariesMatchUnique}\]
\[\langle 1 \rangle\text{ HistorySummariesMatchUnique}(\]
\[\ll1\text{HistorySummary}1, \ll1\text{HistorySummary}2, \ll2\text{HistorySummary}, \text{hashBarrier}) \triangleq\]
\[(\wedge \text{HistorySummariesMatch}(\ll1\text{HistorySummary}1, \ll2\text{HistorySummary}, \text{hashBarrier})\]
\[\wedge \text{HistorySummariesMatch}(\ll1\text{HistorySummary}2, \ll2\text{HistorySummary}, \text{hashBarrier}))\]
\[\Rightarrow \ll1\text{HistorySummary}1 = \ll1\text{HistorySummary}2\]

To prove the universally quantified expression, we take a set of variables of the appropriate types.

\[\langle 1 \rangle\text{. TAKE } \ll1\text{HistorySummary}1, \ll1\text{HistorySummary}2 \in \text{HashType},\]
\[\ll2\text{HistorySummary} \in \text{HistorySummaryType},\]
\[\text{hashBarrier} \in \text{HashType}\]

\[\langle 1 \rangle\ll2\text{InitialHistorySummary} \triangleq [\text{anchor} \mapsto \text{BaseHashValue}, \text{extension} \mapsto \text{BaseHashValue}]\]

Our proof is inductive. First, we prove the base case, which is when the Memoir-Opt history summary equals the initial history summary.

\[\langle 1 \rangle\text{2. } \ll2\text{HistorySummary} = \ll2\text{InitialHistorySummary} \Rightarrow\]
\[\text{HistorySummariesMatchUnique}(\]
\[\ll1\text{HistorySummary}1, \ll1\text{HistorySummary}2, \ll2\text{HistorySummary}, \text{hashBarrier})\]

We assume the antecedent of the base step.

\[\langle 2 \rangle\text{1. HAVE } \ll2\text{HistorySummary} = \ll2\text{InitialHistorySummary}\]

The `HistorySummariesMatchUnique` predicate states an implication. It suffices to assume the antecedent and prove the consequent.

\[\langle 2 \rangle\text{2. SUFFICES}\]
\[\text{ASSUME}\]
\[\wedge \text{HistorySummariesMatch}(\ll1\text{HistorySummary}1, \ll2\text{HistorySummary}, \text{hashBarrier})\]
\[\wedge \text{HistorySummariesMatch}(\ll1\text{HistorySummary}2, \ll2\text{HistorySummary}, \text{hashBarrier})\]
\[\text{PROVE}\]
\[\ll1\text{HistorySummary}1 = \ll1\text{HistorySummary}2\]
\[\text{BY DEF } \text{HistorySummariesMatchUnique}\]

Memoir-Basic history summary 1 equals the base hash value, by the definition of `HistorySummariesMatch` for a Memoir-Opt history summary that equals the initial history summary.

\[\langle 2 \rangle\text{3. } \ll1\text{HistorySummary}1 = \text{BaseHashValue}\]
\[\langle 3 \rangle\text{1. } \text{HistorySummariesMatch}(\ll1\text{HistorySummary}1, \ll2\text{InitialHistorySummary}, \text{hashBarrier}) =\]
\[\ll1\text{HistorySummary}1 = \text{BaseHashValue}\]
\[\langle 4 \rangle\text{1. } \ll2\text{InitialHistorySummary} \in \text{HistorySummaryType}\]
\[\text{BY } \langle 1 \rangle\text{1}, \langle 2 \rangle\text{1}\]
\[\langle 4 \rangle\text{2. QED}\]
\[\text{BY } \langle 1 \rangle\text{1}, \langle 4 \rangle\text{1}, \text{HistorySummariesMatchDefinition}\]
\[\langle 3 \rangle\text{2. } \text{HistorySummariesMatch}(\ll1\text{HistorySummary}1, \ll2\text{InitialHistorySummary}, \text{hashBarrier})\]
\[\text{BY } \langle 2 \rangle\text{1}, \langle 2 \rangle\text{2}\]
\[\langle 3 \rangle\text{3. QED}\]
Memoir-Basic history summary 2 equals the base hash value, by the definition of \textit{HistorySummariesMatch} for a Memoir-Opt history summary that equals the initial history summary.

(2.4) \(ll_1\text{HistorySummary}2 = \text{BaseHashValue}\)

(3.1) \textit{HistorySummariesMatch}(ll_1\text{HistorySummary}2, ll_2\text{InitialHistorySummary}, hashBarrier) =

\(ll_1\text{HistorySummary}2 = \text{BaseHashValue}\)

(4.1) \(ll_2\text{InitialHistorySummary} \in \text{HistorySummaryType}\)

BY (1.1), (2.1)

(4.2) QED

BY (1.1), (4.1), \textit{HistorySummariesMatchDefinition}

(3.2) \textit{HistorySummariesMatch}(ll_1\text{HistorySummary}2, ll_2\text{InitialHistorySummary}, hashBarrier)

BY (2.1), (2.2)

(3.3) QED

BY (3.1), (3.2)

Since the two Memoir-Basic history summaries each equal the base hash value, they equal each other.

(2.5) QED

BY (2.3), (2.4)

Second, we prove the inductive case: If the previous history summaries defined in the \textit{HistorySummariesMatchRecursion} predicate are unique, then their successor history summaries are unique.

(1.3) \(\exists\ input_1, input_2 \in \text{InputType},\)

\(\text{previousLL}_1\text{HistorySummary}1, \text{previousLL}_1\text{HistorySummary}2 \in \text{HashType},\)

\(\text{previousLL}_2\text{HistorySummary} \in \text{HistorySummaryType}:\)

\(\wedge \text{HistorySummariesMatchRecursion}(\)

\(ll_1\text{HistorySummary}1, ll_2\text{HistorySummary}, hashBarrier)!(\)

\(input_1, \text{previousLL}_1\text{HistorySummary}1, \text{previousLL}_2\text{HistorySummary})\)

\(\wedge \text{HistorySummariesMatchRecursion}(\)

\(ll_1\text{HistorySummary}2, ll_2\text{HistorySummary}, hashBarrier)!(\)

\(input_2, \text{previousLL}_1\text{HistorySummary}2, \text{previousLL}_2\text{HistorySummary})\)

\(\wedge \text{HistorySummariesMatchUnique}(\)

\(\text{previousLL}_1\text{HistorySummary}1, \text{previousLL}_1\text{HistorySummary}2, \text{previousLL}_2\text{HistorySummary}, hashBarrier)\)

\(\Rightarrow \)

\(\text{HistorySummariesMatchUnique}(\)

\(ll_1\text{HistorySummary}1, ll_1\text{HistorySummary}2, ll_2\text{HistorySummary}, hashBarrier)\)

To prove the universally quantified expression, we take a set of variables of the appropriate types.

(2.1) TAKE \(input_1, input_2 \in \text{InputType},\)

\(\text{previousLL}_1\text{HistorySummary}1, \text{previousLL}_1\text{HistorySummary}2 \in \text{HashType},\)

\(\text{previousLL}_2\text{HistorySummary} \in \text{HistorySummaryType}\)

We assume the antecedent of the inductive step.

(2.2) HAVE \(\wedge \text{HistorySummariesMatchRecursion}(\)

\(ll_1\text{HistorySummary}1, ll_2\text{HistorySummary}, hashBarrier)!(\)

\(input_1, \text{previousLL}_1\text{HistorySummary}1, \text{previousLL}_2\text{HistorySummary})\)

\(\wedge \text{HistorySummariesMatchRecursion}(\)

\(ll_1\text{HistorySummary}2, ll_2\text{HistorySummary}, hashBarrier)!(\)

\(input_2, \text{previousLL}_1\text{HistorySummary}2, \text{previousLL}_2\text{HistorySummary})\)

\(\wedge \text{HistorySummariesMatchUnique}(\)

\(\text{previousLL}_1\text{HistorySummary}1, \text{previousLL}_1\text{HistorySummary}2, \text{previousLL}_2\text{HistorySummary},\)
The HistorySummariesMatchUnique predicate states an implication. It suffices to assume the antecedent and prove the consequent.

(2)3. SUFFICES

ASSUME

\[ \land \text{HistorySummariesMatch}(ll1\text{HistorySummary}1, ll2\text{HistorySummary}, hashBarrier) \]
\[ \land \text{HistorySummariesMatch}(ll1\text{HistorySummary}2, ll2\text{HistorySummary}, hashBarrier) \]

PROVE

\[ ll1\text{HistorySummary}1 = ll1\text{HistorySummary}2 \]

BY DEF HistorySummariesMatchUnique

We prove that the two inputs from the separate instances of the HistorySummariesMatchRecursion predicate are equal to each other.

(2)4. input1 = input2

(3)1. LL2HistorySummaryIsSuccessor(

\[ ll2\text{HistorySummary}, \text{previousLL2HistorySummary}, \text{input1}, \text{hashBarrier} \]

BY (2)2

We re-state the definitions from LL2HistorySummaryIsSuccessor for input 1.

(3) ll2SuccessorHistorySummary1 ≡ Successor(\text{previousLL2HistorySummary}, \text{input1}, \text{hashBarrier})

(3) ll2CheckpointedSuccessorHistorySummary1 ≡ Checkpoint(ll2SuccessorHistorySummary1)

We hide the definitions.

(3) HIDE DEF ll2SuccessorHistorySummary1, ll2CheckpointedSuccessorHistorySummary1

We re-state the definitions from Successor for input 1.

(3) securedInput1 ≡ Hash(\text{hashBarrier}, \text{input1})

(3) newAnchor ≡ previousLL2HistorySummary.\text{anchor}

(3) newExtension1 ≡ Hash(\text{previousLL2HistorySummary}.\text{extension}, \text{securedInput1})

(3) ll2NewHistorySummary1 \equiv [

\text{anchor} \mapsto \text{newAnchor},
\text{extension} \mapsto \text{newExtension1}]

We prove the types of the definitions from Successor and the definition of ll2SuccessorHistorySummary1, with help from the SuccessorDefsTypeSafeLemma.

(3)2. \land \text{securedInput1} \in \text{HashType}
\land \text{newAnchor} \in \text{HashType}
\land \text{newExtension1} \in \text{HashType}
\land ll2NewHistorySummary1 \in \text{HistorySummaryType}
\land ll2NewHistorySummary1.\text{anchor} \in \text{HashType}
\land ll2NewHistorySummary1.\text{extension} \in \text{HashType}

BY (1)1, (2)1, SuccessorDefsTypeSafeLemma

(3)3. ll2SuccessorHistorySummary1 \in \text{HistorySummaryType}

BY (3)2 DEF ll2SuccessorHistorySummary1, Successor

We hide the definitions.

(3) HIDE DEF securedInput1, newAnchor, newExtension1, ll2NewHistorySummary1

We re-state the definitions from Checkpoint for input 1.

(3) checkpointedAnchor1 ≡ Hash(ll2SuccessorHistorySummary1.\text{anchor}, ll2SuccessorHistorySummary1.\text{extension})

(3) ll2CheckpointedHistorySummary1 ≡ [

\text{anchor} \mapsto \text{checkpointedAnchor1},
\text{extension} \mapsto \text{BaseHashValue}]

We prove the types of the definitions from Checkpoint and the definition of ll2CheckpointedSuccessorHistorySummary1, with help from the CheckpointDefsTypeSafeLemma.

(3)4. \land \text{checkpointedAnchor1} \in \text{HashType}
We hide the definitions.

(3) HIDE DEF checkpointedAnchor1, ll2CheckpointedHistorySummary1

(3)6. LL2HistorySummaryIsSuccessor(  
    ll2HistorySummary, previousLL2HistorySummary, input2, hashBarrier)  
    BY (2)2

We re-state the definitions from LL2HistorySummaryIsSuccessor for input 2.

(3) ll2SuccessorHistorySummary2 ≜ Successor(previousLL2HistorySummary, input2, hashBarrier)  
(3) ll2CheckpointedSuccessorHistorySummary2 ≜ Checkpoint(ll2SuccessorHistorySummary2)

We hide the definitions.

(3) HIDE DEF ll2SuccessorHistorySummary2, ll2CheckpointedSuccessorHistorySummary2

We re-state the definitions from Successor for input 2.

(3) securedInput2 ≜ Hash(hashBarrier, input2)  
(3) newExtension2 ≜ Hash(previousLL2HistorySummary.extension, securedInput2)  
(3) ll2NewHistorySummary2 ≜  
    anchor ↦ newAnchor,  
    extension ↦ newExtension2

We prove the types of the definitions from Successor and the definition of ll2SuccessorHistorySummary2, with help from the SuccessorDefsTypeSafeLemma.

(3)7. ∧ securedInput2 ∈ HashType  
    ∧ newExtension2 ∈ HashType  
    ∧ ll2NewHistorySummary2 ∈ HistorySummaryType  
    ∧ ll2NewHistorySummary2.anchor ∈ HashType  
    ∧ ll2NewHistorySummary2.extension ∈ HashType  
    BY (1)1, (2)1, SuccessorDefsTypeSafeLemma  
    DEF newAnchor

(3)8. ll2SuccessorHistorySummary2 ∈ HistorySummaryType  
    BY (3)7 DEF ll2SuccessorHistorySummary2, Successor, newAnchor

We hide the definitions.

(3) HIDE DEF securedInput2, newExtension2, ll2NewHistorySummary2

We re-state the definitions from Checkpoint for input 2.

(3) checkpointedAnchor2 ≜  
    Hash(ll2SuccessorHistorySummary2.anchor, ll2SuccessorHistorySummary2.extension)  
(3) ll2CheckpointedHistorySummary2 ≜  
    anchor ↦ checkpointedAnchor2,  
    extension ↦ BaseHashValue

We prove the types of the definitions from Checkpoint and the definition of ll2CheckpointedSuccessorHistorySummary2, with help from the CheckpointDefsTypeSafeLemma.

(3)9. ∧ checkpointedAnchor2 ∈ HashType  
    ∧ ll2CheckpointedHistorySummary2 ∈ HistorySummaryType  
    ∧ ll2CheckpointedHistorySummary2.anchor ∈ HashType  
    ∧ ll2CheckpointedHistorySummary2.extension ∈ HashType  
    BY (3)8, CheckpointDefsTypeSafeLemma

(3)10. ll2CheckpointedSuccessorHistorySummary2 ∈ HistorySummaryType  
    BY (3)8, (3)9 DEF ll2CheckpointedSuccessorHistorySummary2, Checkpoint

We hide the definitions.
We prove that the successor history summaries, as defined in the let of the \texttt{LL2HistorySummaryIsSuccessor} predicate, are equal to each other.

We consider two cases. In the first case, the Memoir-Opt history summary has an extension field that equals the base hash value.

The \texttt{LL2HistorySummaryIsSuccessor} predicate expresses a disjunction. An history summary can be a successor either by being a direct successor or by being a checkpoint of a successor. We prove that, in this case, the history summary is a checkpoint of a successor formed with input 1. It cannot be a direct successor because its extension field is the base hash value, and any direct successor will have a non-base extension field.

The same logic as above applies to the the checkpointed successor formed with input 2.

Because the two checkpointed successor history summaries are each equal to the same value, they are equal to each other.
We prove the conclusion: The successor history summaries, as defined in the \texttt{LL2HistorySummaryIsSuccessor} predicate, are equal to each other.

The checkpointed history summaries, as defined in the \texttt{Checkpoint} operator, are equal to each other.

As proven above, the checkpointed history summaries, as defined in the \texttt{LL2HistorySummaryIsSuccessor} predicate, are equal.

The extension field of each successor history summary is not equal to the base hash value, because they are direct successors.

The conclusion follows from the definitions, because the Checkpoint operator is injective under the preimage constraint of an extension field that is unequal to the base hash value.

The individual fields of the successor history summaries are equal to each other, thanks to the collision resistance of the hash function.

Ideally, this \texttt{QED} step should just read:

However, the prover seems to get a little confused in this instance. We make life easier for the prover by defining some local variables and hiding their definitions before appealing to the \texttt{HashCollisionResistant} assumption.
In the second case, the Memoir-Opt history summary has an extension field that does not equal the base hash value. The same logic as above applies to the successor formed with input 2.

\[\begin{align*}
\langle 8\rangle 2. & \quad h_2 \in HashDomain \\
& \text{by } (7) 3 \\
\langle 8\rangle 3. & \quad h_1 \in HashDomain \\
& \text{by } (7) 4 \\
\langle 8\rangle 4. & \quad h_4 \in HashDomain \\
& \text{by } (7) 5 \\
\langle 8\rangle 5. & \quad \text{Hash}(h_1, h_2) = \text{Hash}(h_4, h_2) \\
& \text{by } (7) 1 \text{ DEF checkpointedAnchor1, checkpointedAnchor2} \\
\langle 8\rangle 6. & \quad h_1 = h_1 \land h_4 = h_4 \\
& \text{(9) hide DEF h_1, h_2, h_4, h_1, h_2} \\
& \text{(9) QED} \\
& \text{by } (8) 1, (8) 2, (8) 3, (8) 4, (8) 5, HashCollisionResistant \\
\langle 8\rangle 7. & \quad \text{QED} \\
& \text{by } (8) 6
\end{align*}\]

Because the fields are equal, the records are equal, but proving this requires that we prove the types of the records and invoke the HistorySummaryRecordCompositionLemma.

\[\begin{align*}
\langle 6\rangle 3. & \quad \text{ll2SuccessorHistorySummary} 1 \in HistorySummaryType \\
& \text{by } (3) 3 \\
\langle 6\rangle 4. & \quad \text{ll2SuccessorHistorySummary} 2 \in HistorySummaryType \\
& \text{by } (3) 8 \\
\langle 6\rangle 5. & \quad \text{QED} \\
& \text{by } (6) 2, (6) 3, (6) 4, HistorySummaryRecordCompositionLemma \text{DEF HistorySummaryType}
\end{align*}\]

In the second case, the Memoir-Opt history summary has an extension field that does not equal the base hash value.

(4) 2. \text{CASE ll2HistorySummary.extension} \neq \text{BaseHashValue} 

The LL2HistorySummaryIsSuccessor predicate expresses a disjunction. An history summary can be a successor either by being a direct successor or by being a checkpoint of a successor. We prove that, in this case, history summary 1 is a direct successor formed with input 1. It cannot be a checkpoint because its extension field is not the base hash value, and any checkpoint will have a base extension field.

\[\begin{align*}
\langle 5\rangle 1. & \quad \text{ll2HistorySummary} = \text{ll2SuccessorHistorySummary} 1 \\
\langle 6\rangle 1. & \quad \text{ll2HistorySummary} = \text{ll2SuccessorHistorySummary} 1 \\
& \lor \text{ll2HistorySummary} = \text{ll2CheckpointedSuccessorHistorySummary} 1 \\
\langle 7\rangle 1. & \quad \text{LL2HistorySummaryIsSuccessor}(
\text{ll2HistorySummary, previousLL2HistorySummary, input1, hashBarrier}) \\
& \text{by } (2) 2 \\
\langle 7\rangle 2. & \quad \text{QED} \\
& \text{by } (7) 1 \\
& \text{DEF LL2HistorySummaryIsSuccessor, ll2SuccessorHistorySummary1,} \\
& \text{ll2CheckpointedSuccessorHistorySummary1}
\end{align*}\]

\[\begin{align*}
\langle 6\rangle 2. & \quad \text{ll2HistorySummary} \neq \text{ll2CheckpointedSuccessorHistorySummary} 1 \\
\langle 7\rangle 1. & \quad \text{ll2CheckpointedSuccessorHistorySummary} 1.\text{extension} = \text{BaseHashValue} \\
& \text{by } (3) 3, \text{CheckpointHasBaseExtensionLemma} \\
& \text{DEF ll2CheckpointedSuccessorHistorySummary1} \\
\langle 7\rangle 2. & \quad \text{QED} \\
& \text{by } (4) 2, (7) 1 \\
\langle 6\rangle 3. & \quad \text{QED} \\
& \text{by } (6) 1, (6) 2
\end{align*}\]

The same logic as above applies to the the successor formed with input 2.

\[\begin{align*}
\langle 5\rangle 2. & \quad \text{ll2HistorySummary} = \text{ll2SuccessorHistorySummary} 2 \\
\langle 6\rangle 1. & \quad \text{ll2HistorySummary} = \text{ll2SuccessorHistorySummary} 2 \\
& \lor \text{ll2HistorySummary} = \text{ll2CheckpointedSuccessorHistorySummary} 2 \\
\langle 7\rangle 1. & \quad \text{LL2HistorySummaryIsSuccessor}(
\text{ll2HistorySummary, previousLL2HistorySummary, input2, hashBarrier}) \\
& \text{by } (2) 2 \\
\langle 7\rangle 2. & \quad \text{QED} \\
\end{align*}\]
\textit{ll2HistorySummary, previousLL2HistorySummary, input2, hashBarrier})

\begin{itemize}
  \item By (2)2
  \item (7)2. QED
  \item By (7)1
  \item \textbf{DEF} \textit{ll2HistorySummaryIsSuccessor, ll2SuccessorHistorySummary2, ll2CheckpointedSuccessorHistorySummary2}
  \item (6)2. \textit{ll2HistorySummary} \neq \textit{ll2CheckpointedSuccessorHistorySummary2}
  \item (7)1. \textit{ll2CheckpointedSuccessorHistorySummary2}.extension = \textit{BaseHashValue}
  \item By (3)8, CheckpointHasBaseExtensionLemma
  \item \textbf{DEF} \textit{ll2CheckpointedSuccessorHistorySummary2}
  \item (7)2. QED
  \item By (4)2, (7)1
  \item (6)3. QED
  \item By (6)1, (6)2
\end{itemize}

Because the two successor history summaries are each equal to the same value, they are equal to each other.

\begin{itemize}
  \item By (5)1, (5)2
\end{itemize}

The two cases are exhaustive.

\begin{itemize}
  \item By (4)1, (4)2
\end{itemize}

The secured inputs are equal to each other, because the new extensions are equal to each other, and the hash is collision-resistant.

\begin{itemize}
  \item By (4)1
  \item (4)2. \textit{newExtension1} = \textit{newExtension2}
  \item (5)1. \textit{ll2NewHistorySummary1} = \textit{ll2NewHistorySummary2}
  \item By (3)11
  \item \textbf{DEF} \textit{ll2SuccessorHistorySummary1, ll2SuccessorHistorySummary2}. Successor,
    \textit{ll2NewHistorySummary1, ll2NewHistorySummary2, newExtension1, newExtension2, newAnchor, securedInput1, securedInput2}
    \item By (5)1 DEF \textit{ll2NewHistorySummary1, ll2NewHistorySummary2}
  \item (4)2. \textit{previousLL2HistorySummary}.extension \in \textit{HashDomain}
  \item By (2)1 DEF \textit{HistorySummaryType, HashDomain}
  \item (4)3. \textit{securedInput1} \in \textit{HashDomain}
  \item By (3)2 DEF \textit{HashDomain}
  \item (4)4. \textit{securedInput2} \in \textit{HashDomain}
  \item By (3)7 DEF \textit{HashDomain}
  \item (4)5. QED
\end{itemize}

Ideally, this QED step should just read:

\begin{itemize}
  \item By (4)1, (4)2, (4)3, (4)4, HashCollisionResistant
\end{itemize}

However, the prover seems to get a little confused in this instance. We make life easier for the prover by defining some local variables and hiding their definitions before appealing to the \textit{HashCollisionResistant} assumption.

\begin{itemize}
  \item (5) h1a \triangleq previousLL2HistorySummary.extension
  \item (5) h2a \triangleq securedInput1
  \item (5) h2b \triangleq securedInput2
  \item (5)1. h1a \in HashDomain
    \item By (4)2
  \item (5)2. h2a \in HashDomain
    \item By (4)3
  \item (5)3. h2b \in HashDomain
    \item By (4)4
\end{itemize}

243
(5)4. \( \text{Hash}(h_1a, h_2a) = \text{Hash}(h_1a, h_2b) \)
   \text{by (4)1 def newExtension1, newExtension2}

(5)5. \( h_2a = h_2b \)

(6)\ HIDE \ def \ h_1a, h_2a, h_2b

(6)1. QED
   \text{by (5)1, (5)2, (5)3, (5)4, HashCollisionResistant}

(5)6. QED
   \text{by (5)5}

The input values are equal to each other, because the secured inputs are equal, and the hash function is collision-resistant.

(3)16. QED
   (4)1. \text{securedInput1} = \text{securedInput2}
   \text{by (3)12}

(4)2. \text{hashBarrier} \in \text{HashDomain}
   \text{by (1)1 def HashDomain}

(4)3. \text{input1} \in \text{HashDomain}
   \text{by (2)1 def HashDomain}

(4)4. \text{input2} \in \text{HashDomain}
   \text{by (2)2 def HashDomain}

(4)5. QED
   \text{by (4)1, (4)2, (4)3, (4)4, HashCollisionResistant}
   \text{def securedInput1, securedInput2}

We prove that the two previous Memoir-Basic history summaries from the separate instances of the \text{HistorySummariesMatchRecursion} predicate are equal to each other. This follows from the recursive use of the \text{HistorySummariesMatchUnique} predicate on the previous history summaries.

(2)5. \( \text{previousLL1HistorySummary1} = \text{previousLL1HistorySummary2} \)
   (3)1. \text{HistorySummariesMatch}(
      \text{previousLL1HistorySummary1, previousLL2HistorySummary, hashBarrier})
      \text{by (2)2 def HistorySummariesMatchRecursion}

(3)2. \text{HistorySummariesMatch}(
      \text{previousLL1HistorySummary2, previousLL2HistorySummary, hashBarrier})
      \text{by (2)2 def HistorySummariesMatchRecursion}

(3)3. \text{HistorySummariesMatchUnique}(
      \text{previousLL1HistorySummary1,}
      \text{previousLL1HistorySummary2,}
      \text{previousLL2HistorySummary,}
      \text{hashBarrier})
      \text{by (2)2}

(3)4. QED
   \text{by (3)1, (3)2, (3)3 def HistorySummariesMatchUnique}

The conclusion follows directly from the equality of the previous history summaries and the inputs, since the current history summaries are generated as hashes of those values.

(2)6. QED
   (3)1. \( \text{ll1HistorySummary1} = \text{Hash}(\text{previousLL1HistorySummary1, input1}) \)
      \text{by (2)2 def HistorySummariesMatchRecursion}

(3)2. \( \text{ll1HistorySummary2} = \text{Hash}(\text{previousLL1HistorySummary2, input2}) \)
      \text{by (2)2 def HistorySummariesMatchRecursion}

(3)3. QED
   \text{by (2)4, (2)5, (3)1, (3)2}

Due to language limitations, we do not state or prove the step that ties together the base case and the inductive case into a proof for all cases.

(1)4. QED
The **AuthenticatorsMatchUniqueLemma** asserts that there is a unique Memoir-Basic authenticator that matches a particular Memoir-Opt authenticator.

**THEOREM AuthenticatorsMatchUniqueLemma**

\[∀ \ll1\text{Authenticator}1, \ll1\text{Authenticator}2 ∈ \text{MACType},\]
\[\ll2\text{Authenticator} ∈ \text{MACType},\]
\[\text{symmetricKey}1, \text{symmetricKey}2 ∈ \text{SymmetricKeyType},\]
\[\text{hashBarrier} ∈ \text{HashType} :\]
\[\begin{align*}
∀ & \text{AuthenticatorsMatch}(\ll1\text{Authenticator}1, \ll2\text{Authenticator}, \text{symmetricKey}1, \text{hashBarrier}) \\
& \text{AuthenticatorsMatch}(\ll1\text{Authenticator}2, \ll2\text{Authenticator}, \text{symmetricKey}2, \text{hashBarrier})
\end{align*}\]
\[⇒ \ll1\text{Authenticator}1 = \ll1\text{Authenticator}2\]

To prove the universally quantified expression, we take a set of variables in the appropriate types.

\(\langle 1 \rangle 1. \text{TAEK}\) \ll1\text{Authenticator}1, \ll1\text{Authenticator}2 ∈ \text{MACType},
\ll2\text{Authenticator} ∈ \text{MACType},
\text{symmetricKey}1, \text{symmetricKey}2 ∈ \text{SymmetricKeyType},
\text{hashBarrier} ∈ \text{HashType}

We assume the antecedent of the implication.

\(\langle 1 \rangle 2. \text{HAVE}\) \[\begin{align*}
& \text{AuthenticatorsMatch}(\ll1\text{Authenticator}1, \ll2\text{Authenticator}, \text{symmetricKey}1, \text{hashBarrier}) \\
& \text{AuthenticatorsMatch}(\ll1\text{Authenticator}2, \ll2\text{Authenticator}, \text{symmetricKey}2, \text{hashBarrier})
\end{align*}\]

We pick a set of variables that witness the existentials inside the **AuthenticatorsMatch** predicate for authenticator 1.

\(\langle 1 \rangle 3. \text{PICK}\) \[\begin{align*}
& \text{stateHash}1 ∈ \text{HashType},
& \ll1\text{HistorySummary}1 ∈ \text{HashType},
& \ll2\text{HistorySummary}1 ∈ \text{HistorySummaryType} :\]
\[\text{AuthenticatorsMatch}(\ll1\text{Authenticator}1, \ll2\text{Authenticator}, \text{symmetricKey}1, \text{hashBarrier})!\]
\[\text{stateHash}1, \ll1\text{HistorySummary}1, \ll2\text{HistorySummary}1\)

BY \(\langle 1 \rangle 2\) DEF **AuthenticatorsMatch**

We re-state the definitions from the \textit{let} in **AuthenticatorsMatch** for authenticator 1.

\(\langle 1 \rangle 4. \text{LET}\) \[\begin{align*}
& \ll1\text{HistoryStateBinding}1 \triangleq \text{Hash}(\ll1\text{HistorySummary}1, \text{stateHash}1) \\
& \ll2\text{HistorySummaryHash}1 \triangleq \text{Hash}(\ll2\text{HistorySummary}1.\text{anchor}, \ll2\text{HistorySummary}1.\text{extension}) \\
& \ll2\text{HistoryStateBinding}1 \triangleq \text{Hash}(\ll2\text{HistorySummaryHash}1, \text{stateHash}1)
\end{align*}\]

We prove the types of the definitions, with the help of the **AuthenticatorsMatchDefsTypeSafeLemma**.

\(\langle 1 \rangle 4.\) \[\begin{align*}
& \ll1\text{HistoryStateBinding}1 ∈ \text{HashType} \\
& \ll2\text{HistorySummaryHash}1 ∈ \text{HashType} \\
& \ll2\text{HistoryStateBinding}1 ∈ \text{HashType}
\end{align*}\]

BY \(\langle 1 \rangle 3, \text{AuthenticatorsMatchDefsTypeSafeLemma}\)

We hide the definitions.

\(\langle 1 \rangle \text{HIDE DEF}\) \ll1\text{HistoryStateBinding}1, \ll2\text{HistorySummaryHash}1, \ll2\text{HistoryStateBinding}1

We pick a set of variables that witness the existentials inside the **AuthenticatorsMatch** predicate for authenticator 2.

\(\langle 1 \rangle 5. \text{PICK}\) \[\begin{align*}
& \text{stateHash}2 ∈ \text{HashType},
& \ll1\text{HistorySummary}2 ∈ \text{HashType},
& \ll2\text{HistorySummary}2 ∈ \text{HistorySummaryType}:
\end{align*}\]
Because the Memoir-Opt history summary hashes are equal, the Memoir-Opt history summaries are equal. 

Next, we prove that the two Memoir-Opt history summary hashes are equal and the two state hashes are equal. This follows from the collision resistance of the MAC functions applied to the previous step, followed by the collision resistance of the hash function.

Next, we prove that the two Memoir-Opt history summary hashes are equal and the two state hashes are equal. This follows from the collision resistance of the MAC functions applied to the previous step, followed by the collision resistance of the hash function.
follows from the unforgeability of the MAC are generated with identical symmetric keys and identical history state bindings. The equality of the symmetric keys finally, we prove the equality of the Memoir-Basic state authenticators. This equality follows from the fact that they

⟨ HistorySummariesMatchUniqueLemma ⟩

Memoir-Basic history summaries follows from the equality of the Memoir-Opt history summaries, by employing the equality of the Memoir-Basic history summaries and the equality of the state hashes. And the equality of the

10.

by

⟨ HashCollisionResistant ⟩

However, the prover seems to get a little confused in this instance. We make life easier for the prover by defining some local variables and hiding their definitions before appealing to the HashCollisionResistant assumption.

⟨ HistorySummaryRecordCompositionLemma ⟩

Finally, we prove the equality of the Memoir-Basic state authenticators. This equality follows from the fact that they are generated with identical symmetric keys and identical history state bindings. The equality of the symmetric keys follows from the unforgeability of the MAC. The equality of the Memoir-Basic history state bindings follows from the equality of the Memoir-Basic history summaries and the equality of the state hashes. And the equality of the Memoir-Basic history summaries follows from the equality of the Memoir-Opt history summaries, by employing the HistorySummariesMatchUniqueLemma.

10. QED

2. ll1HistoryStateBinding1 = ll1HistoryStateBinding2

3. ll1HistorySummary1 = ll1HistorySummary2

4. HistorySummariesMatch(ll1HistorySummary1, ll2HistorySummary1, hashBarrier)

5. QED

6. HistorySummariesMatch(ll1HistorySummary2, ll2HistorySummary2, hashBarrier)

7. QED

8. HistorySummariesMatchUniqueLemma

9. QED

10. QED

2. ll1HistoryStateBinding1 = ll1HistoryStateBinding2

3. ll1HistorySummary1 = ll1HistorySummary2

4. HistorySummariesMatch(ll1HistorySummary1, ll2HistorySummary1, hashBarrier)

5. QED

6. HistorySummariesMatch(ll1HistorySummary2, ll2HistorySummary2, hashBarrier)

7. QED

8. HistorySummariesMatchUniqueLemma

9. QED

2. QED
The AuthenticatorSetsMatchUniqueLemma asserts that there is a unique Memoir-Basic set of authenticators that matches a particular Memoir-Opt set of authenticators.

THEOREM AuthenticatorSetsMatchUniqueLemma \( \triangleq \)
\[
\forall ll1AuthenticatorSet1, ll1AuthenticatorSet2 \in \text{subset MACType}, 
\quad ll2AuthenticatorSet \in \text{subset MACType}, 
\quad \text{symmetricKey} \in \text{SymmetricKeyType}, 
\quad \text{hashBarrier} \in \text{HashType} : 
\quad (\land \text{AuthenticatorSetsMatch}(ll1AuthenticatorSet1, ll2AuthenticatorSet, \text{symmetricKey}, \text{hashBarrier}) 
\land \text{AuthenticatorSetsMatch}(ll1AuthenticatorSet2, ll2AuthenticatorSet, \text{symmetricKey}, \text{hashBarrier})) 
\Rightarrow ll1AuthenticatorSet1 = ll1AuthenticatorSet2
\]

To prove the universally quantified expression, we take a set of variables in the appropriate types.

\( \langle 1 \rangle. \text{T}AKE ll1AuthenticatorSet1, ll1AuthenticatorSet2 \in \text{subset MACType}, 
ll2AuthenticatorSet \in \text{subset MACType}, 
\text{symmetricKey} \in \text{SymmetricKeyType}, 
\text{hashBarrier} \in \text{HashType} 
\)

We assume the antecedent of the implication.

\( \langle 2 \rangle. \text{H}AVE \land \text{AuthenticatorSetsMatch}(ll1AuthenticatorSet1, ll2AuthenticatorSet, \text{symmetricKey}, \text{hashBarrier}) 
\land \text{AuthenticatorSetsMatch}(ll1AuthenticatorSet2, ll2AuthenticatorSet, \text{symmetricKey}, \text{hashBarrier}) \)

The proof has two main steps, which are mirror images of each other. First, we prove that every Memoir-Basic authenticator in set 1 is also in set 2.

\( \langle 3 \rangle. \forall ll1Authenticator \in ll1AuthenticatorSet1 : 
ll1Authenticator \in ll1AuthenticatorSet2 
\)

To prove the universally quantified expression, we take an arbitrary authenticator in Memoir-Basic set 1.

\( \langle 4 \rangle. \text{T}AKE ll1Authenticator1 \in ll1AuthenticatorSet1 
\)

Then we pick a matching authenticator in the Memoir-Opt set.

\( \langle 5 \rangle. \text{P}ICK ll2Authenticator \in ll2AuthenticatorSet : 
\text{AuthenticatorsMatch}(ll1Authenticator1, ll2Authenticator, \text{symmetricKey}, \text{hashBarrier}) 
\)

We then pick a matching authenticator in set 2.

\( \langle 6 \rangle. \text{P}ICK ll1Authenticator2 \in ll1AuthenticatorSet2 : 
\text{AuthenticatorsMatch}(ll1Authenticator2, ll2Authenticator, \text{symmetricKey}, \text{hashBarrier}) 
\)
by (1)2 def AuthenticatorSetsMatch
The two Memoir-Basic authenticators match, by the AuthenticatorsMatchUniqueLemma.

(2)4. \( \text{ll1Authenticator}_1 = \text{ll1Authenticator}_2 \)
(3)1. \( \text{ll1Authenticator}_1 \in \text{MACType} \)
    by (1)1, (2)1
(3)2. \( \text{ll1Authenticator}_2 \in \text{MACType} \)
    by (1)1, (2)3
(3)3. \( \text{ll2Authenticator} \in \text{MACType} \)
    by (1)1, (2)2
(3)4. QED
    by (2)2, (2)3, (3)1, (3)2, (3)3, AuthenticatorsMatchUniqueLemma
(2)5. QED
    by (2)3, (2)4

Second, we prove that every Memoir-Basic authenticator in set 2 is also in set 1.

(1)4. \( \forall \text{ll1Authenticator} \in \text{ll1AuthenticatorSet}_2 : \)
    \( \text{ll1Authenticator} \in \text{ll1AuthenticatorSet}_1 \)
To prove the universally quantified expression, we take an arbitrary authenticator in Memoir-Basic set 2.

(2)2. PICK \( \text{ll2Authenticator} \in \text{ll2AuthenticatorSet} : \)
    AuthenticatorsMatch(\( \text{ll1Authenticator}_2, \text{ll2Authenticator}, \text{symmetricKey}, \text{hashBarrier} \))
    by (1)2 def AuthenticatorSetsMatch
We then pick a matching authenticator in set 1.

(2)3. PICK \( \text{ll1Authenticator}_1 \in \text{ll1AuthenticatorSet}_1 : \)
    AuthenticatorsMatch(\( \text{ll1Authenticator}_1, \text{ll2Authenticator}, \text{symmetricKey}, \text{hashBarrier} \))
    by (1)2 def AuthenticatorSetsMatch
The two Memoir-Basic authenticators match, by the AuthenticatorsMatchUniqueLemma.

(2)4. \( \text{ll1Authenticator}_1 = \text{ll1Authenticator}_2 \)
(3)1. \( \text{ll1Authenticator}_1 \in \text{MACType} \)
    by (1)1, (2)3
(3)2. \( \text{ll1Authenticator}_2 \in \text{MACType} \)
    by (1)1, (2)1
(3)3. \( \text{ll2Authenticator} \in \text{MACType} \)
    by (1)1, (2)2
(3)4. QED
    by (2)2, (2)3, (3)1, (3)2, (3)3, AuthenticatorsMatchUniqueLemma
(2)5. QED
    by (2)3, (2)4
(1)5. QED
    by (1)3, (1)4

The \( \text{LL2NVRAMLogicalHistorySummaryTypeSafe} \) lemma asserts that the \( \text{LL2NVRAMLogicalHistorySummary} \) is of \( \text{HistorySummaryType} \) as long as the Memoir-Opt type invariant is satisfied.

THEOREM \( \text{LL2NVRAMLogicalHistorySummaryTypeSafe} \triangleq \)
\( \land \text{LL2TypeInvariant} \Rightarrow \text{LL2NVRAMLogicalHistorySummary} \in \text{HistorySummaryType} \)
\( \land \text{LL2TypeInvariant'} \Rightarrow \text{LL2NVRAMLogicalHistorySummary'} \in \text{HistorySummaryType} \)
The AuthenticatorInSetLemma states that if (1) two state authenticators match across the two specs, (2) two authenticator sets match across the two specs, and (3) in the Memoir-Opt spec, the authenticator is an element of the authenticator set, then in the Memoir-Basic spec, the authenticator is an element of the authenticator set.

**Theorem** AuthenticatorInSetLemma \(\triangleq\)
\[
\forall \ ll1\text{Authenticator} \in \text{MACType}, \\
ll2\text{Authenticator} \in \text{MACType}, \\
ll1\text{Authenticators} \in \text{subset MACType}, \\
ll2\text{Authenticators} \in \text{subset MACType}, \\
symmetricKey1 \in \text{SymmetricKeyType}, \\
symmetricKey2 \in \text{SymmetricKeyType}, \\
hashBarrier \in \text{HashType}:
\]
\[
(\land \text{AuthenticatorsMatch}(\ll1\text{Authenticator}, \ll2\text{Authenticator}, \text{symmetricKey1}, \text{hashBarrier}) \\
\land \text{AuthenticatorSetsMatch}(\ll1\text{Authenticators}, \ll2\text{Authenticators}, \text{symmetricKey2}, \text{hashBarrier}) \\
\land \ll2\text{Authenticator} \in \ll2\text{Authenticators}) \\
\Rightarrow \ll1\text{Authenticator} \in \ll1\text{Authenticators}
\]

To prove the universally quantified expression, we take a set of variables of the appropriate types.

\(\langle 1\rangle 1.\) TAKE \(\ll1\text{Authenticator}1 \in \text{MACType},\) \\
\(\ll2\text{Authenticator}1 \in \text{MACType},\) \\
\(\ll1\text{Authenticators} \in \text{subset MACType},\) \\
\(\ll2\text{Authenticators} \in \text{subset MACType},\)
symmetricKey\(_1\) ∈ SymmetricKeyType,
symmetricKey\(_2\) ∈ SymmetricKeyType,
hashBarrier ∈ HashType

To prove the implication, we assume the antecedent.

\(\langle 1\rangle\). \textbf{have} \quad ∧ \quad \textbf{AuthenticatorsMatch}(
\quad ll1\text{Authenticator}\(_1\), ll2\text{Authenticator}\(_1\), symmetricKey\(_1\), hashBarrier)
\quad ∧ \quad \textbf{AuthenticatorSetsMatch}(
\quad ll1\text{Authenticator}, ll2\text{Authenticator}, symmetricKey\(_2\), hashBarrier)
\quad ∧ \quad ll2\text{Authenticator}\(_1\) ∈ ll2\text{Authenticators}

We pick a new Memoir-Basic authenticator that (1) is in the given Memoir-Basic authenticator set and (2) matches the given Memoir-Opt authenticator.

\(\langle 1\rangle\). \textbf{pick} \ll1\text{Authenticator}\(_2\) ∈ ll1\text{Authenticators}:
\quad \textbf{AuthenticatorsMatch}(
\quad ll1\text{Authenticator}\(_2\), ll2\text{Authenticator}\(_1\), symmetricKey\(_2\), hashBarrier)

We first prove that such an element exists.

\(\langle 2\rangle\). \exists ll1\text{Authenticator}\(_2\) ∈ ll1\text{Authenticators}:
\quad \textbf{AuthenticatorsMatch}(
\quad ll1\text{Authenticator}\(_2\), ll2\text{Authenticator}\(_1\), symmetricKey\(_2\), hashBarrier)

By hypothesis of the lemma, the given Memoir-Opt authenticator is an element of the given Memoir-Opt authenticator set.

\(\langle 3\rangle\). ll2\text{Authenticator}\(_1\) ∈ ll2\text{Authenticators}
\quad \textbf{by} \quad \langle 1\rangle\)

Because the two authenticator sets match, there exists a matching Memoir-Basic authenticator for every Memoir-Opt authenticator. This follows from the definition of AuthenticatorSetsMatch.

\(\langle 3\rangle\). \forall ll2\text{Authenticator}\(_2\) ∈ ll2\text{Authenticators}:
\quad \exists ll1\text{Authenticator}\(_2\) ∈ ll1\text{Authenticators}:
\quad \textbf{AuthenticatorsMatch}(
\quad ll1\text{Authenticator}\(_2\), ll2\text{Authenticator}\(_2\), symmetricKey\(_2\), hashBarrier)
\quad \textbf{by} \quad \langle 1\rangle\)

\(\langle 4\rangle\). AuthenticatorSetsMatch(
\quad ll1\text{Authenticator}, ll2\text{Authenticator}, symmetricKey\(_2\), hashBarrier)
\quad \textbf{by} \quad \langle 1\rangle\)

Therefore, there exists a matching Memoir-Basic authenticator for the given Memoir-Opt authenticator.

\(\langle 3\rangle\). QED
\quad \textbf{by} \quad \langle 3\rangle\), \langle 3\rangle\)

Because such an element exists, we can pick it.

\(\langle 2\rangle\). QED
\quad \textbf{by} \quad \langle 2\rangle\)

We prove that the given Memoir-Basic authenticator equals the newly picked Memoir-Basic authenticator.

\(\langle 1\rangle\). ll1\text{Authenticator}\(_1\) = ll1\text{Authenticator}\(_2\)

The given Memoir-Basic authenticator matches the given Memoir-Opt authenticator, by hypothesis of the lemma.

\(\langle 2\rangle\). AuthenticatorsMatch(
\quad ll1\text{Authenticator}\(_1\), ll2\text{Authenticator}\(_1\), symmetricKey\(_1\), hashBarrier)
\quad \textbf{by} \quad \langle 1\rangle\)

The newly picked Memoir-Basic authenticator matches the given Memoir-Opt authenticator, by property of the pick.

\(\langle 2\rangle\). AuthenticatorsMatch(
\quad ll1\text{Authenticator}\(_2\), ll2\text{Authenticator}\(_1\), symmetricKey\(_2\), hashBarrier)
\quad \textbf{by} \quad \langle 1\rangle\)
Since both Memoir-Basic authenticators match the given Memoir-Opt authenticator, the \textit{AuthenticatorsMatchUniqueLemma} tells us that they must be equal.

(2). \text{QED}

We first have to prove some types for the \textit{AuthenticatorsMatchUniqueLemma}.

(3).1. $ll_1\text{Authenticator} \in MACType$
   \hspace{1em} \text{by (1)1}

(3).2. $ll_1\text{Authenticator} \in MACType$
   \hspace{1em} (4).1. $ll_1\text{Authenticator} \in ll_1\text{Authenticators}$
   \hspace{1em} \text{by (1)3}

(4).2. $ll_1\text{Authenticators} \subseteq \text{subset} \ MACType$
   \hspace{1em} \text{by (1)1}

(4).3. \text{QED}

\hspace{1em} \text{by (4)1, (4)2}

(3).3. $ll_2\text{Authenticator} \in MACType$
   \hspace{1em} \text{by (1)1}

(3).4. $\text{symmetricKey} \in \text{SymmetricKeyType}$
   \hspace{1em} \text{by (1)1}

(3).5. $\text{symmetricKey} \in \text{SymmetricKeyType}$
   \hspace{1em} \text{by (1)1}

(3).6. $\text{hashBarrier} \in \text{HashType}$
   \hspace{1em} \text{by (1)1}

Then we can apply the \textit{AuthenticatorsMatchUniqueLemma} directly.

(3).7. \text{QED}

\hspace{1em} \text{by (2)1, (2)2, (3)1, (3)2, (3)3, (3)4, (3)5, (3)6, AuthenticatorsMatchUniqueLemma}

The newly picked Memoir-Basic authenticator is an element of the given Memoir-Opt authenticator set, by property of the pick.

(1)5. $ll_1\text{Authenticator} \in ll_1\text{Authenticators}$
   \hspace{1em} \text{by (1)3}

Since the two Memoir-Basic authenticators are equal, and the newly picked authenticator is an element of the given Memoir-Opt authenticator set, it follows that the given authenticator is an element of the given Memoir-Opt authenticator set.

(1)6. \text{QED}

\hspace{1em} \text{by (1)4, (1)5}

The \textit{AuthenticatorGeneratedLemma} states that if (1) two inputs summaries match across the two specs, (2) two authenticators match across the two specs, and (3) in the Memoir-Opt spec, the authenticator is generated as a MAC for the history state binding formed from the history summary and any state hash, then in the Memoir-Basic spec, the authenticator is generated as a MAC for the history state binding formed from the history summary and the same state hash.

\textbf{THEOREM AuthenticatorGeneratedLemma} \hspace{1em} \triangleq

\[ \forall \text{stateHash} \in \text{HashType}, \]

\[ ll_1\text{HistorySummary} \in \text{HashType}, \]

\[ ll_2\text{HistorySummary} \in \text{HistorySummaryType}, \]

\[ ll_1\text{Authenticator} \in \text{MACType}, \]

\[ ll_2\text{Authenticator} \in \text{MACType}, \]

\[ \text{symmetricKey} \in \text{SymmetricKeyType}, \]

\[ \text{hashBarrier} \in \text{HashType} : \]
We pick a set of variables that satisfy the existentially quantified variables in $\text{AuthenticatorsMatch}$. To prove the implication, it suffices to assume the antecedent and prove the consequent. We hide the definitions. We prove the types of the definitions, with help from the $\text{AuthenticatorsMatchDefsTypeSafeLemma}$. We re-state the definitions from the $\text{LET}$ in the lemma. We prove the types of the definitions. We hide the definitions. To prove the implication, it suffices to assume the antecedent and prove the consequent. We pick a set of variables that satisfy the existentially quantified variables in $\text{AuthenticatorsMatch}$. We prove that such a set of variables exists. This follows because $\text{AuthenticatorsMatch}$ is assumed by the lemma.
\( \text{ll1HistorySummary2} \in \text{HashType}, \)
\( \text{ll2HistorySummary2} \in \text{HistorySummaryType} : \)
\begin{align*}
\text{AuthenticatorsMatch}( & \\
\text{ll1Authenticator, ll2Authenticator, symmetricKey1, hashBarrier}) ! ( & \\
\text{stateHash2, ll1HistorySummary2, ll2HistorySummary2}) & \\
\text{(3)1. AuthenticatorsMatch}( & \\
\text{ll1Authenticator, ll2Authenticator, symmetricKey1, hashBarrier}) & \\
\text{BY (1)3} & \\
\text{(3)2. QED} & \\
\text{BY (3)1 DEF AuthenticatorsMatch} & \\
\text{(2)2. QED} & \\
\text{BY (2)1} & \\
\end{align*}

We re-state the definitions from the \text{LET} in \text{AuthenticatorsMatch}.

\( (1) \text{ll1HistoryStateBinding2} \triangleq \text{Hash}(\text{ll1HistorySummary2, stateHash2}) \)
\( (1) \text{ll2HistorySummaryHash2} \triangleq \text{Hash}(\text{ll2HistorySummary2.anchor, ll2HistorySummary2.extension}) \)
\( (1) \text{ll2HistoryStateBinding2} \triangleq \text{Hash}(\text{ll2HistorySummaryHash2, stateHash2}) \)

We prove the types of the definitions, with help from the \text{AuthenticatorsMatchDefsTypeSafeLemma}.

\( (1)5. \wedge \text{ll1HistoryStateBinding2} \in \text{HashType} & \\
\wedge \text{ll2HistorySummaryHash2} \in \text{HashType} & \\
\wedge \text{ll2HistoryStateBinding2} \in \text{HashType} \)
\text{BY (1)4, AuthenticatorsMatchDefsTypeSafeLemma}

We hide the definitions.

\( (1) \text{HIDE DEF ll1HistoryStateBinding2, ll2HistorySummaryHash2, ll2HistoryStateBinding2} \)

The biggest outer step of the proof is showing that in the Memoir-Opt spec, the given values of history summary and state hash equal the newly picked values of history summary and state hash.

\( (1)6. \wedge \text{ll2HistorySummary1} = \text{ll2HistorySummary2} & \\
\wedge \text{stateHash1} = \text{stateHash2} \)

In the Memoir-Opt spec, the given values of history summary hash and state hash equal the newly picked values of history summary hash and state hash.

\( (2)1. \wedge \text{ll2HistorySummaryHash1} = \text{ll2HistorySummaryHash2} & \\
\wedge \text{stateHash1} = \text{stateHash2} \)

In the Memoir-Opt spec, the given history state binding equals the newly picked history state binding. This follows from the collision resistance of the \text{MAC}.

\( (3)1. \text{ll2HistoryStateBinding1} = \text{ll2HistoryStateBinding2} \)
\( (4)1. \text{ll2Authenticator} = \text{GenerateMAC(symmetricKey2, ll2HistoryStateBinding1)} \)
\text{BY (1)3}
\( (4)2. \text{ValidateMAC(symmetricKey1, ll2HistoryStateBinding2, ll2Authenticator)} \)
\text{BY (1)4 DEF ll2HistoryStateBinding2, ll2HistorySummaryHash2}
\( (4)3. \text{QED} \)
\( (5)1. \text{symmetricKey1} \in \text{SymmetricKeyType} \)
\text{BY (1)1}
\( (5)2. \text{symmetricKey2} \in \text{SymmetricKeyType} \)
\text{BY (1)1}
\( (5)3. \text{ll2HistoryStateBinding1} \in \text{HashType} \)
\text{BY (1)2}
\( (5)4. \text{ll2HistoryStateBinding2} \in \text{HashType} \)
\text{BY (1)5}
\( (5)5. \text{QED} \)
\text{BY (4)1, (4)2, (5)1, (5)2, (5)3, (5)4, MACCollisionResistant}

The history summary hashes and state hashes are each respectively equal, because the hash is collision resistant.

\( (3)2. \text{QED} \)
In the Memoir-Opt spec, the given value of history summary equals the newly picked value of history summary.

The corresponding fields of the two history summaries (the given one and the newly picked one) are equal, because the hash is collision resistant.
Ideally, this QED step should just read:

\begin{align*}
\text{BY (4)1, (4)2, (4)3, (4)4, (4)5, } & \text{HashCollisionResistant} \text{ DEF } ll2HistorySummaryHash1, ll2HistorySummaryHash2 \\
\text{However, the prover seems to get a little confused in this instance. We make life easier for the prover by defining}\ & \text{some local variables and hiding their definitions before appealing to the } HashCollisionResistant \text{ assumption.} \\
(5) & h1a \triangleq ll2HistorySummary1.\text{anchor} \\
(5) & h2a \triangleq ll2HistorySummary1.\text{extension} \\
(5) & h1b \triangleq ll2HistorySummary2.\text{anchor} \\
(5) & h2b \triangleq ll2HistorySummary2.\text{extension} \\
(5)1. & \text{Hash}(h1a, h2a) = \text{Hash}(h1b, h2b) \\
\text{BY (4)1 DEF } ll2HistorySummaryHash1, ll2HistorySummaryHash2 \\
(5)2. & h1a \in HashDomain \\
\text{BY (4)2} \\
(5)3. & h2a \in HashDomain \\
\text{BY (4)3} \\
(5)4. & h1b \in HashDomain \\
\text{BY (4)4} \\
(5)5. & h2b \in HashDomain \\
\text{BY (4)5} \\
(5)6. & h1a = h1b \land h2a = h2b \\
(6) & \text{HIDE DEF } h1a, h2a, h1b, h2b \\
(6)1. & \text{QED} \\
\text{BY (5)1, (5)2, (5)3, (5)4, (5)5, } & \text{HashCollisionResistant} \\
(5)7. & \text{QED} \\
\text{BY (5)6} \\
\end{align*}

Because the fields are equal, the records are equal, but proving this requires that we prove the types of the records and invoke the HistorySummaryRecordCompositionLemma.

\begin{align*}
(3)2. & ll2HistorySummary1 \in HistorySummaryType \\
\text{BY (1)1} \\
(3)3. & ll2HistorySummary2 \in HistorySummaryType \\
\text{BY (1)4} \\
(3)4. & \text{QED} \\
\text{BY (3)1, (3)2, (3)3, HistorySummaryRecordCompositionLemma}\text{DEF } HistorySummaryType \\
\end{align*}

For clarity, we restate one of the conjuncts above.

\begin{align*}
(2)3. & stateHash1 = stateHash2 \\
\text{BY (2)1} \\
(2)4. & \text{QED} \\
\text{BY (2)2, (2)3} \\
\end{align*}

We then prove that in the Memoir-Basic spec, the given value of history summary equals the newly picked value of history summary. This follows from the HistorySummariesMatchUniqueLemma.

\begin{align*}
(1)7. & ll1HistorySummary1 = ll1HistorySummary2 \\
(2)1. & \text{HistorySummariesMatch}(ll1HistorySummary1, ll2HistorySummary1, hashBarrier) \\
\text{BY (1)3} \\
(2)2. & \text{HistorySummariesMatch}(ll1HistorySummary2, ll2HistorySummary2, hashBarrier) \\
\text{BY (1)4} \\
(2)3. & ll1HistorySummary1 \in HashType \\
\text{BY (1)1} \\
(2)4. & ll1HistorySummary2 \in HashType \\
\text{BY (1)4} \\
(2)5. & ll2HistorySummary1 \in HistorySummaryType \\
\text{BY (1)1} \\
\end{align*}

256
In the final step, we show in the Memoir-Basic spec that (1) the given value of the history state binding equals the newly picked value of the history state binding, (2) the two symmetric keys are equal, and (3) the given authenticator is generated as a MAC of the newly picked history state binding. This directly implies that the given authenticator is generated as a MAC of the given history state binding, which is the goal of the lemma.

(1). QED

(2). l1HistoryStateBinding1 = l1HistoryStateBinding2

(3). stateHash1 = stateHash2

BY (1)7

(3). QED

BY (3)1, (3)2 DEF l1HistoryStateBinding1, l1HistoryStateBinding2

(2). symmetricKey1 = symmetricKey2

(3). l2Authenticator = GenerateMAC(symmetricKey2, l2HistoryStateBinding1)

BY (1)3

(3). ValidateMAC(symmetricKey1, l2HistoryStateBinding2, l2Authenticator)

BY (1)4 DEF l2HistoryStateBinding2, l2HistorySummaryHash2

(3). QED

(4). symmetricKey1 ∈ SymmetricKeyType

BY (1)1

(4). symmetricKey2 ∈ SymmetricKeyType

BY (1)1

(4). l2HistoryStateBinding1 ∈ HashType

BY (1)2

(4). l2HistoryStateBinding2 ∈ HashType

BY (1)5

(4). QED

BY (3)1, (3)2, (4)1, (4)2, (4)3, (4)4, MACUnforgeable

(2). l1Authenticator = GenerateMAC(symmetricKey1, l1HistoryStateBinding2)

BY (1)4, (1)6 DEF l1HistoryStateBinding2

(2). QED

BY (2)1, (2)2, (2)3 DEF l1HistoryStateBinding2

The AuthenticatorValidatedLemma states that if (1) two inputs summaries match across the two specs, (2) two authenticators match across the two specs, and (3) in the Memoir-Opt spec, the authenticator is a valid MAC for the history state binding formed from the history summary and any state hash, then in the Memoir-Basic spec, the authenticator is a valid MAC for the history state binding formed from the history summary and the same state hash.

THEOREM AuthenticatorValidatedLemma \( \Delta \)

\( \forall \) stateHash ∈ HashType,

l1HistorySummary ∈ HashType,

l2HistorySummary ∈ HistorySummaryType,

l1Authenticator ∈ MACType,

l2Authenticator ∈ MACType,

symmetricKey1 ∈ SymmetricKeyType,

symmetricKey2 ∈ SymmetricKeyType,
hashBarrier ∈ HashType:

LET
ll1HistoryStateBinding ≜ Hash(ll1HistorySummary, stateHash)
ll2HistorySummaryHash ≜ Hash(ll2HistorySummary, anchor, ll2HistorySummary.extension)
ll2HistoryStateBinding ≜ Hash(ll2HistorySummaryHash, stateHash)

IN
(∧ HistorySummariesMatch(ll1HistorySummary, ll2HistorySummary, hashBarrier)
∧ AuthenticatorsMatch(ll1Authenticator, ll2Authenticator, symmetricKey1, hashBarrier)
∧ ValidateMAC(symmetricKey2, ll2HistoryStateBinding, ll2Authenticator))
⇒ ValidateMAC(symmetricKey2, ll1HistoryStateBinding, ll1Authenticator)

To prove the universally quantified expression, we take a set of variables of the appropriate types.

{1}1. TAKE stateHash ∈ HashType,
   ll1HistorySummary ∈ HashType,
   ll2HistorySummary ∈ HistorySummaryType,
   ll1Authenticator ∈ MACType,
   ll2Authenticator ∈ MACType,
   symmetricKey1 ∈ SymmetricKeyType,
   symmetricKey2 ∈ SymmetricKeyType,
   hashBarrier ∈ HashType

We re-state the definitions from the LET in the lemma.

{1}1 ll1HistoryStateBinding ≜ Hash(ll1HistorySummary, stateHash)
{1}1 ll2HistorySummaryHash ≜ Hash(ll2HistorySummary, anchor, ll2HistorySummary.extension)
{1}1 ll2HistoryStateBinding ≜ Hash(ll2HistorySummaryHash, stateHash)

We prove the types of the definitions, with help from the AuthenticatorsMatchDefsTypeSafeLemma.

{1}2. ∧ ll1HistoryStateBinding ∈ HashType
   ∧ ll2HistorySummaryHash ∈ HashType
   ∧ ll2HistoryStateBinding ∈ HashType
   BY {1}1, AuthenticatorsMatchDefsTypeSafeLemma

We hide the definitions.

{1}1 HIDE DEF ll1HistoryStateBinding, ll2HistorySummaryHash, ll2HistoryStateBinding

To prove the implication, it suffices to assume the antecedent and prove the consequent.

{1}3. SUFFICES
   ASSUME
   (∧ HistorySummariesMatch(ll1HistorySummary, ll2HistorySummary, hashBarrier)
   ∧ AuthenticatorsMatch(ll1Authenticator, ll2Authenticator, symmetricKey1, hashBarrier)
   ∧ ValidateMAC(symmetricKey2, ll2HistoryStateBinding, ll2Authenticator))
   PROVE
   ValidateMAC(symmetricKey2, ll1HistoryStateBinding, ll1Authenticator)
   BY DEF ll1HistoryStateBinding, ll2HistorySummaryHash, ll2HistoryStateBinding

Using the MACConsistent property, we show that, since the Memoir-Opt authenticator is a valid MAC of the Memoir-Opt history state binding, it must have been generated as a MAC of this history state binding.

{1}4. ll2Authenticator = GenerateMAC(symmetricKey2, ll2HistoryStateBinding)
{1}2. ValidateMAC(symmetricKey2, ll2HistoryStateBinding, ll2Authenticator)
   BY {1}3
{1}2. symmetricKey1 ∈ SymmetricKeyType
   BY {1}1
{1}2. symmetricKey2 ∈ SymmetricKeyType
   BY {1}1
Then, using the AuthenticatorGeneratedLemma, we show that the Memoir-Basic authenticator is generated as a MAC of the Memoir-Basic history state binding.

\[\text{AuthenticatorGeneratedLemma} \triangleq \]

\[
\begin{align*}
\forall \text{symmetricKey1, symmetricKey2, hashBarrier} \\
\text{AuthenticatorGeneratedLemma}(\text{stateHash, ll1HistorySummary, ll2HistorySummary, ll1Authenticator, ll2Authenticator, symmetricKey1, symmetricKey2, hashBarrier) !} & \tag{3.1} \\
\end{align*}
\]

Finally, using the MACComplete property, we show that the Memoir-Basic authenticator is a valid MAC of the Memoir-Basic history state binding.

\[\text{MACComplete} \triangleq \]

\[
\begin{align*}
\forall \text{symmetricKey2} \\
\text{MACComplete}(\text{ll1HistoryStateBinding, ll2HistorySummaryHash, ll2HistoryStateBinding}) & \tag{3.6} \\
\end{align*}
\]

The HistorySummariesMatchAcrossCheckpointLemma asserts that for any pair of Memoir-Opt history summaries for which one is a checkpoint of the other, there is a unique Memoir-Basic history summary that matches either Memoir-Opt history summary.

\[\text{THEOREM HistorySummariesMatchAcrossCheckpointLemma} \triangleq \]
∀ ll1HistorySummary1, ll1HistorySummary2 ∈ HashType,
ll2HistorySummary1, ll2HistorySummary2 ∈ HistorySummaryType,
hashBarrier ∈ HashType :
( ∨ HistorySummariesMatch(ll1HistorySummary1, ll2HistorySummary1, hashBarrier)
∧ HistorySummariesMatch(ll1HistorySummary2, ll2HistorySummary2, hashBarrier)
∧ ll2HistorySummary2 = Checkpoint(ll2HistorySummary1))
⇒ ll1HistorySummary1 = ll1HistorySummary2

To prove the universally quantified expression, we take a set of variables of the appropriate types.

We assume the antecedent.

There are two separate cases, one for the base case of the HistorySummariesMatch predicate and one for the recursion. The base case is trivial.

We'll first pick values for the existential variables inside the HistorySummariesMatchRecursion predicate that satisfy the predicate for the 1 variables in the antecedent of the theorem. We know the HistorySummariesMatchRecursion predicate holds, because the Memoir-Opt history summary does not equal the initial history summary, according to this case.

DEF HistorySummariesMatchRecursion
We’ll next pick values for the existential variables inside the \textit{HistorySummariesMatchRecursion} predicate that satisfy the predicate for the 2 variables in the antecedent of the theorem. We know the \textit{HistorySummariesMatchRecursion} predicate holds, because the Memoir-Opt history summary does not equal the initial history summary, but proving this latter point is slightly involved.

(2) \text{Pick } \texttt{input2}\in\texttt{InputType},

\begin{align*}
\texttt{ll1PreviousHistorySummary2} \in \texttt{HashType}, \\
\texttt{ll2PreviousHistorySummary2} \in \texttt{HistorySummaryType} : \\
\textit{HistorySummariesMatchRecursion}( \\
\texttt{ll1HistorySummary2, ll2HistorySummary2, hashBarrier})! ( \\
\texttt{input2, ll1PreviousHistorySummary2, ll2PreviousHistorySummary2})
\end{align*}

(3) \texttt{ll1HistorySummary2} \in \texttt{HashType} \\
\land \texttt{ll2HistorySummary2} \in \texttt{HistorySummaryType} \\
\land \texttt{hashBarrier} \in \texttt{HashType}

By (1)1

(3) \texttt{HistorySummariesMatch}(\texttt{ll1HistorySummary2, ll2HistorySummary2, hashBarrier})

By (1)2

(3) \texttt{ll2HistorySummary2} \neq \texttt{ll2InitialHistorySummary}

Inputs summary 2 does not equal the initial history summary because its anchor field does not equal the base hash value.

(4) \texttt{ll2HistorySummary2.anchor} \neq \texttt{BaseHashValue}

We use two more levels of case analysis. For the first level, we show that at least one of the two fields in history summary 1 is not equal to the base hash value.

(5) \texttt{ll2HistorySummary1.extension} \neq \texttt{BaseHashValue}

\begin{align*}
\texttt{ll2HistorySummary2.anchor} & = \\
\text{Hash}(&\texttt{ll2HistorySummary1.anchor, ll2HistorySummary1.extension})
\end{align*}

(7) \texttt{ll2HistorySummary2} = \texttt{Checkpoint}(\texttt{ll2HistorySummary1})

By (1)2

(7) QED

By (5)2, (7)1 DEF Checkpoint

(6) \texttt{ll2HistorySummary1.anchor} \in \texttt{HashDomain}

By (1)1 DEF \texttt{HistorySummaryType}, \texttt{HashDomain}

(6) \texttt{ll2HistorySummary1.extension} \in \texttt{HashDomain}

By (1)1 DEF \texttt{HistorySummaryType}, \texttt{HashDomain}

(6) QED

By (6)1, (6)2, (6)3, \texttt{BaseHashValueUnique}

We consider the case in which the anchor field of history summary 1 is not equal to the base hash value. We consider two sub-cases.

(5)3. \texttt{ll2HistorySummary1.anchor} \neq \texttt{BaseHashValue}

In the first sub-case, the extension field of history summary 1 is also not equal to the base hash value. We already proved this case above.

(6)1. CASE \texttt{ll2HistorySummary1.extension} \neq \texttt{BaseHashValue}

By (5)2, (6)1
We prove that the Memoir-Opt previous history summaries are equal, and the inputs are equal.

(6)\textbf{2. CASE \(ll2\text{HistorySummary1}.\text{extension} = \text{BaseHashValue}\)}
\textbf{3. \(ll2\text{HistorySummary2}.\text{anchor} = ll2\text{HistorySummary1}.\text{anchor}\)}
\textbf{4. \(ll2\text{HistorySummary2} = \text{Checkpoint}(ll2\text{HistorySummary1})\)}
\textbf{BY (1)\textbf{2}}
\textbf{5. (8)\textbf{2. QED}}
\textbf{6. (8)\textbf{1 DEF \textbf{Checkpoint}})
\textbf{7. (7)\textbf{2. QED}}
\textbf{8. (5)\textbf{3, (7)\textbf{1}}}
\textbf{9. (6)\textbf{3. QED}}
\textbf{10. (6)\textbf{1, (6)\textbf{2}}}
\textbf{11. (5)\textbf{4. QED}}
\textbf{12. (5)\textbf{1, (5)\textbf{2, (5)\textbf{3}}}
\textbf{13. (4)\textbf{2. \(ll2\text{InitialHistorySummary}.\text{anchor} = \text{BaseHashValue}\)}}
\textbf{14. \textbf{OBSVIOUS}}
\textbf{15. (4)\textbf{3. QED}}
\textbf{16. (4)\textbf{1, (4)\textbf{2}}}
\textbf{17. (3)\textbf{4. QED}}
\textbf{18. (3)\textbf{1, (3)\textbf{2, (3)\textbf{3. HistorySummariesMatchDefinition}}}}
\textbf{19. \textbf{DEF HistorySummariesMatchRecursion}}

We prove that the Memoir-Opt previous history summaries are equal, and the inputs are equal.

\textbf{(2)}\textbf{3. \& \(ll2\text{PreviousHistorySummary1} = ll2\text{PreviousHistorySummary2}\)}
\textbf{\& \(\text{input1} = \text{input2}\)}
\textbf{(2)\textbf{3. Several useful facts follow directly from the theorem’s assumptions and the above picks.}}
\textbf{(3)\textbf{1. \(LL2\text{HistorySummaryIsSuccessor}(\)}}
\textbf{\textbf{\& \(ll2\text{HistorySummary1}, ll2\text{PreviousHistorySummary1}, \text{input1}, \text{hashBarrier})\)}}
\textbf{\& \(\text{input1})\text{.}}
\textbf{\textbf{(3)\textbf{2. \(LL2\text{HistorySummaryIsSuccessor}(\)}}
\textbf{\textbf{\& \(ll2\text{HistorySummary2}, ll2\text{PreviousHistorySummary2}, \text{input2}, \text{hashBarrier})\)}}
\textbf{\& \(\text{input2})\text{.}}
\textbf{\textbf{(3)\textbf{3. \(ll2\text{HistorySummary2} = \text{Checkpoint}(ll2\text{HistorySummary1})\)}}}
\textbf{\textbf{(3)\textbf{3. \textbf{We re-state the definitions from \textbf{LL2HistorySummaryIsSuccessor} for history summary 1.}}}}
\textbf{\textbf{(3)\textbf{3. \textbf{We hide the definitions.}}}}
\textbf{\textbf{(3)\textbf{3. \textbf{We re-state the definitions from \textbf{Successor} for history summary 1.}}}}
\textbf{\textbf{(3)\textbf{3. \textbf{We prove the types of the definitions from \textbf{Successor} and the definition of \textbf{ll2SuccessorHistorySummary1}, with help from the \textbf{SuccessorDefsTypeSafeLemma}.}}}}

\textbf{(3)\textbf{4. \& \(\text{securedInput1} \in \text{Hash Type}\)}
\textbf{\& \(\text{newAnchor1} \in \text{Hash Type}\)}
\textbf{\& \(\text{newExtension1} \in \text{Hash Type}\)}

262
\[ l2\text{NewHistorySummary}1 \in \text{HistorySummaryType} \]
\[ l2\text{NewHistorySummary}1.\text{anchor} \in \text{HashType} \]
\[ l2\text{NewHistorySummary}1.\text{extension} \in \text{HashType} \]
\[ \text{BY (1) 1, (2) 1, SuccessorDefsTypeSafeLemma} \]

(3.5) \( l2\text{SuccessorHistorySummary}1 \in \text{HistorySummaryType} \)
\[ \text{BY (3) 4 DEF } l2\text{SuccessorHistorySummary}1, \text{Successor} \]

We hide the definitions.

(3) \text{HIDE DEF } \text{securedInput1, newAnchor1, newExtension1, } l2\text{NewHistorySummary}1

We re-state the definitions from \text{Checkpoint} for history summary 1.

(3) \text{checkpointedAnchor1 } \triangleq \text{Hash}(l2\text{SuccessorHistorySummary}1.\text{anchor}, l2\text{SuccessorHistorySummary}1.\text{extension})

(3) \text{l2CheckpointedHistorySummary1 } \triangleq \lfloor \text{anchor } \mapsto \text{checkpointedAnchor1},

\text{extension } \mapsto \text{BaseHashValue} \rfloor

We prove the types of the definitions from \text{Checkpoint} and the definition of \text{l2CheckpointedSuccessorHistorySummary1}, with help from the \text{CheckpointDefsTypeSafeLemma}.

(3.6) \wedge \text{checkpointedAnchor1 } \in \text{HashType}

\wedge \text{l2CheckpointedHistorySummary1 } \in \text{HistorySummaryType}

\wedge \text{l2CheckpointedHistorySummary1.\text{anchor} } \in \text{HashType}

\wedge \text{l2CheckpointedHistorySummary1.\text{extension} } \in \text{HashType}

\[ \text{BY (3) 5, CheckpointDefsTypeSafeLemma} \]

(3.7) \text{l2CheckpointedSuccessorHistorySummary1 } \in \text{HistorySummaryType}
\[ \text{BY (3) 5, (3) 6 DEF } l2\text{CheckpointedSuccessorHistorySummary1, Checkpoint} \]

We hide the definitions.

(3) \text{HIDE DEF } \text{checkpointedAnchor1, } l2\text{CheckpointedHistorySummary1}

We re-state the definitions from \text{L2HistorySummaryIsSuccessor} for history summary 2.

(3) \text{l2SuccessorHistorySummary2 } \triangleq \text{Successor}(l2\text{PreviousHistorySummary2}, \text{input2, hashBarrier})

(3) \text{l2CheckpointedSuccessorHistorySummary2 } \triangleq \text{Checkpoint}(l2\text{SuccessorHistorySummary2})

We hide the definitions.

(3) \text{HIDE DEF } l2\text{SuccessorHistorySummary2}, l2\text{CheckpointedSuccessorHistorySummary2}

We re-state the definitions from \text{Successor} for history summary 2.

(3) \text{securedInput2 } \triangleq \text{Hash}(\text{hashBarrier}, \text{input2})

(3) \text{newAnchor2 } \triangleq l2\text{PreviousHistorySummary2.\text{anchor}}

(3) \text{newExtension2 } \triangleq \text{Hash}(l2\text{PreviousHistorySummary2.\text{extension}, securedInput2})

(3) \text{l2NewHistorySummary2 } \triangleq \lfloor \text{anchor } \mapsto \text{newAnchor2},

\text{extension } \mapsto \text{newExtension2} \rfloor

We prove the types of the definitions from \text{Successor} and the definition of \text{l2SuccessorHistorySummary2}, with help from the \text{SuccessorDefsTypeSafeLemma}.

(3.8) \wedge \text{securedInput2 } \in \text{HashType}

\wedge \text{newAnchor2 } \in \text{HashType}

\wedge \text{newExtension2 } \in \text{HashType}

\wedge \text{l2NewHistorySummary2 } \in \text{HistorySummaryType}

\wedge \text{l2NewHistorySummary2.\text{anchor} } \in \text{HashType}

\wedge \text{l2NewHistorySummary2.\text{extension} } \in \text{HashType}

\[ \text{BY (1) 1, (2) 1, SuccessorDefsTypeSafeLemma} \]

(3.9) \text{l2SuccessorHistorySummary2 } \in \text{HistorySummaryType}
\[ \text{BY (3) 8 DEF } l2\text{SuccessorHistorySummary2, Successor} \]

We hide the definitions.

(3) \text{HIDE DEF } \text{securedInput2, newAnchor2, newExtension2, l2NewHistorySummary2}

263
We re-state the definitions from Checkpoint for history summary 2.

(3)  \text{checkpointedAnchor}^2 = \text{Hash}(\text{ll}2\text{SuccessorHistorySummary}^2.\text{anchor}, \text{ll}2\text{SuccessorHistorySummary}^2.\text{extension})

(3)  \text{ll}2\text{CheckpointedHistorySummary}^2 = [\text{checkpointedAnchor}^2, \text{extension} \rightarrow \text{BaseHashValue}]

We prove the types of the definitions from Checkpoint and the definition of \text{ll}2\text{CheckpointedSuccessorHistorySummary}^2, with help from the CheckpointDefsTypeSafeLemma.

(3) 10.  \&  \text{checkpointedAnchor}^2 \in \text{HashType}

(3) 11.  \text{ll}2\text{CheckpointedSuccessorHistorySummary}^2 \in \text{HistorySummaryType}

(3) 12.  \text{ll}2\text{CheckpointedSuccessorHistorySummary}^1 = \text{ll}2\text{CheckpointedSuccessorHistorySummary}^2

We hide the definitions.

(3) \text{HIDE DEF checkpointedAnchor}^2, \text{ll}2\text{CheckpointedHistorySummary}^2

The most critical part of the proof is showing that the checkpointed successor history summaries, as defined in the \text{ll}2\text{HistorySummaryIsSuccessor} predicate, are equal. This requires separately considering the cases of whether the extension field of \text{ll}2\text{HistorySummary} equals the base hash value.

(3) 12.  \text{ll}2\text{CheckpointedSuccessorHistorySummary}^1 = \text{ll}2\text{CheckpointedSuccessorHistorySummary}^2

In the first case, the extension field of \text{ll}2\text{HistorySummary} equals the base hash value.

(4) 1.  \text{CASE ll}2\text{HistorySummary}^1.\text{extension} = \text{BaseHashValue}

The \text{ll}2\text{HistorySummaryIsSuccessor} predicate expresses a disjunction. An history summary can be a successor either by being a direct successor or by being a checkpoint of a successor. We prove that, in this case, history summary 1 is a checkpoint of a successor. It cannot be a direct successor because its extension field is the base hash value, and any direct successor will have a non-base extension field.

(5) 1.  \text{ll}2\text{HistorySummary}^1 = \text{ll}2\text{CheckpointedSuccessorHistorySummary}^1

(6) 1.  \lor  \text{ll}2\text{HistorySummary}^1 = \text{ll}2\text{SuccessorHistorySummary}^1

(6) 2.  \text{ll}2\text{HistorySummary}^1 \neq \text{ll}2\text{SuccessorHistorySummary}^1

(7) 1.  \text{ll}2\text{SuccessorHistorySummary}^1.\text{extension} \neq \text{BaseHashValue}

(7) 2.  \text{QED}

(6) 3.  \text{QED}

By (6) 1, (6) 2

Inputs summary 2 is also a checkpoint of a successor, by the same argument as above for history summary 1. The one twist is that we need to prove that the its extension field is the base hash value, which follows from the fact that - when history summary 1 has a base extension field, history summaries 1 and 2 are equal.

(5) 2.  \text{ll}2\text{HistorySummary}^2 = \text{ll}2\text{CheckpointedSuccessorHistorySummary}^2

(6) 1.  \lor  \text{ll}2\text{HistorySummary}^2 = \text{ll}2\text{SuccessorHistorySummary}^2

(6) 2.  \text{ll}2\text{HistorySummary}^2 \neq \text{ll}2\text{SuccessorHistorySummary}^2

(7) 1.  \text{ll}2\text{SuccessorHistorySummary}^2.\text{extension} \neq \text{BaseHashValue}
By (1)1, (2)2, SuccessorHasNonBaseExtensionLemma \( \text{def } \ll 2 \text{SuccessorHistorySummary2} \)

(7)2. QED

(8)1. \( \ll 2 \text{HistorySummary2}. \text{extension} = \text{BaseHashValue} \)

(9)1. \( \ll 2 \text{HistorySummary1} = \ll 2 \text{HistorySummary2} \)

By (3)3, (4)1 DEF Checkpoint

(9)2. QED

By (4)1, (9)1

(8)2. QED

By (7)1, (8)1

(6)3. QED

By (6)1, (6)2

The conclusion follows from the above-proven equalities, given the fact that, when history summary 1 has a base extension field, the Checkpoint operator acts as an identity operator, so history summaries 1 and 2 are equal.

(5)3. QED

(6)1. \( \ll 2 \text{HistorySummary1} = \ll 2 \text{HistorySummary2} \)

By (3)3, (4)1 DEF Checkpoint

(6)2. QED

By (5)1, (5)2, (6)1

In the second case, the extension field of \( \ll 2 \text{HistorySummary} \) does not equal the base hash value.

(4)2. CASE \( \ll 2 \text{HistorySummary1}. \text{extension} \neq \text{BaseHashValue} \)

The \( \ll 2 \text{HistorySummaryIsSuccessor} \) predicate expresses a disjunction. An history summary can be a successor either by being a direct successor or by being a checkpoint of a successor. We prove that, in this case, history summary 1 is a direct successor. It cannot be a checkpoint because its extension field is not the base hash value, and any checkpoint will have a base extension field.

(5)1. \( \ll 2 \text{HistorySummary1} = \ll 2 \text{SuccessorHistorySummary1} \)

(6)1. \( \lor \ll 2 \text{HistorySummary1} = \ll 2 \text{SuccessorHistorySummary1} \)

(6)2. \( \lor \ll 2 \text{CheckpointedSuccessorHistorySummary1} \)

By (3)1

DEF \( \ll 2 \text{HistorySummaryIsSuccessor}, \ll 2 \text{SuccessorHistorySummary1}, \)

(7)1. \( \ll 2 \text{CheckpointedSuccessorHistorySummary1}. \text{extension} = \text{BaseHashValue} \)

By (3)5, CheckpointHasBaseExtensionLemma

DEF \( \ll 2 \text{CheckpointedSuccessorHistorySummary1} \)

(7)2. QED

By (4)2, (7)1

(6)3. QED

By (6)1, (6)2

Inputs summary 2 is a checkpoint of a successor, by the reverse of the above argument for history summary 1. We know that the extension field of history summary 2 must be the base hash value, because it is a checkpoint of history summary 1.

(5)2. \( \ll 2 \text{HistorySummary2} = \ll 2 \text{CheckpointedSuccessorHistorySummary2} \)

(6)1. \( \lor \ll 2 \text{HistorySummary2} = \ll 2 \text{SuccessorHistorySummary2} \)

(6)2. \( \lor \ll 2 \text{CheckpointedSuccessorHistorySummary2} \)

By (3)2

DEF \( \ll 2 \text{HistorySummaryIsSuccessor}, \ll 2 \text{SuccessorHistorySummary2}, \)

(7)1. \( \ll 2 \text{SuccessorHistorySummary2}. \text{extension} \neq \text{BaseHashValue} \)

By (1)1, (2)2, SuccessorHasNonBaseExtensionLemma \( \text{def } \ll 2 \text{SuccessorHistorySummary2} \)

(7)2. \( \ll 2 \text{HistorySummary2}. \text{extension} = \text{BaseHashValue} \)
The conclusion follows from the above-proven equalities, given the parallel construction of checkpoint from history summary in both the history summaries in the antecedent of the theorem and the history summaries defined within the LET of the \textit{LL2HistorySummaryIsSuccessor} predicate.

(5) 3. QED
(6) 1. \textit{ll2CheckpointedSuccessorHistorySummary1} = \textit{Checkpoint(\textit{ll2HistorySummary1})}

By (5) 1 DEF \textit{ll2CheckpointedSuccessorHistorySummary1}
(6) 2. \textit{ll2HistorySummary2} = \textit{Checkpoint(\textit{ll2HistorySummary1})}

By (1) 2
(6) 3. \textit{ll2CheckpointedSuccessorHistorySummary2} = \textit{ll2HistorySummary2}

By (5) 2
(6) 4. QED

By (6) 1, (6) 2, (6) 3

The two cases are exhaustive.

(4) 3. QED

By (4) 1, (4) 2

We prove that the successor history summaries, as defined in the LET of the \textit{LL2HistorySummaryIsSuccessor} predicate, are equal to each other.

(3) 13. \textit{ll2SuccessorHistorySummary1} = \textit{ll2SuccessorHistorySummary2}

The checkpointed history summaries, as defined in the LET of the \textit{Checkpoint} operator, are equal to each other.

(4) 1. \textit{ll2CheckpointedHistorySummary1} = \textit{ll2CheckpointedHistorySummary2}

As proven above by case analysis, the checkpointed inputs summaries, as defined in the LET of the \textit{LL2HistorySummaryIsSuccessor} predicate, are equal.

(5) 1. \textit{ll2CheckpointedSuccessorHistorySummary1} = \textit{ll2CheckpointedSuccessorHistorySummary2}

By (3) 12

The extension field of each successor history summary is not equal to the base hash value, because they are direct successors.

(5) 2. \textit{ll2SuccessorHistorySummary1}.extension \neq \textit{BaseHashValue}

By (1) 1, (2) 1, \textit{SuccessorHasNonBaseExtensionLemma} DEF \textit{ll2SuccessorHistorySummary1}

(5) 3. \textit{ll2SuccessorHistorySummary2}.extension \neq \textit{BaseHashValue}

By (1) 1, (2) 2, \textit{SuccessorHasNonBaseExtensionLemma} DEF \textit{ll2SuccessorHistorySummary2}

The conclusion follows from the definitions, because the Checkpoint operator is injective under the preimage constraint of an extension field that is unequal to the base hash value.

(5) 4. QED

By (5) 1, (5) 2, (5) 3

DEF \textit{ll2CheckpointedSuccessorHistorySummary1}, \textit{ll2CheckpointedSuccessorHistorySummary2},
\textit{Checkpoint}, \textit{ll2CheckpointedHistorySummary1}, \textit{ll2CheckpointedHistorySummary2},
\textit{checkpointedAnchor1}, \textit{checkpointedAnchor2}

The individual fields of the successor history summaries are equal to each other, thanks to the collision resistance of the hash function.

(4) 2. \textit{\&}\; \textit{ll2SuccessorHistorySummary1}.anchor = \textit{ll2SuccessorHistorySummary2}.anchor

\textit{\&}\; \textit{ll2SuccessorHistorySummary1}.extension = \textit{ll2SuccessorHistorySummary2}.extension

(5) 1. \textit{checkpointedAnchor1} = \textit{checkpointedAnchor2}

By (4) 1 DEF \textit{ll2CheckpointedHistorySummary1}, \textit{ll2CheckpointedHistorySummary2}

(5) 2. \textit{ll2SuccessorHistorySummary1}.anchor \in \textit{HashDomain}

By (3) 5 DEF \textit{HistorySummaryType}, \textit{HashDomain}

(5) 3. \textit{ll2SuccessorHistorySummary1}.extension \in \textit{HashDomain}

266
BY (3), 5 DEF HistorySummaryType, HashDomain
(5), 4. ll2SuccessorHistorySummary2.anchor ∈ HashDomain
BY (3), 9 DEF HistorySummaryType, HashDomain
(5), 5. ll2SuccessorHistorySummary2.extension ∈ HashDomain
BY (3), 9 DEF HistorySummaryType, HashDomain
(5), 6. QED

Ideally, this QED step should just read:

BY (5), 1, (5), 2, (5), 3, (5), 4, (5), 5, HashCollisionResistant DEF checkpointedAnchor1, checkpointedAnchor2

However, the prover seems to get a little confused in this instance. We make life easier for the prover by defining some local variables and hiding their definitions before appealing to the HashCollisionResistant assumption.

(6) h1a ≜ ll2SuccessorHistorySummary1.anchor
(6) h2a ≜ ll2SuccessorHistorySummary1.extension
(6) h1b ≜ ll2SuccessorHistorySummary2.anchor
(6) h2b ≜ ll2SuccessorHistorySummary2.extension
(6) 1. h1a ∈ HashDomain
BY (5), 2
(6) 2. h2a ∈ HashDomain
BY (5), 3
(6) 3. h1b ∈ HashDomain
BY (5), 4
(6) 4. h2b ∈ HashDomain
BY (5), 5
(6) 5. Hash(h1a, h2a) = Hash(h1b, h2b)
BY (5), 1 DEF checkpointedAnchor1, checkpointedAnchor2
(6) 6. h1a = h1b ∧ h2a = h2b
(7) HIDE def h1a, h2a, h1b, h2b
(7) 1. QED
BY (6) 1, (6), 2, (6), 3, (6), 4, (6), 5, HashCollisionResistant
(6) 7. QED
BY (6), 6

Because the fields are equal, the records are equal, but proving this requires that we prove the types of the records and invoke the HistorySummaryRecordCompositionLemma.

(4), 3 ll2SuccessorHistorySummary1 ∈ HistorySummaryType
BY (3), 5
(4), 4 ll2SuccessorHistorySummary2 ∈ HistorySummaryType
BY (3), 9
(4), 5. QED
BY (4), 2, (4), 3, (4), 4, HistorySummaryRecordCompositionLemma DEF HistorySummaryType

The values that are hashed to produce the new extension value within the Successor operator are equal to each other, because the new extensions are equal to each other, and the hash is collision-resistant.

(3), 14 ∧ ll2PreviousHistorySummary1.extension = ll2PreviousHistorySummary2.extension ∧ securedInput1 = securedInput2
(4), 1. newExtension1 = newExtension2
(5), 1. ll2NewHistorySummary1 = ll2NewHistorySummary2
BY (3), 13
DEF ll2SuccessorHistorySummary1, ll2SuccessorHistorySummary2, Successor,
ll2NewHistorySummary1, ll2NewHistorySummary2, newExtension1, newExtension2,
newAnchor1, newAnchor2, securedInput1, securedInput2
(5), 2. QED
BY (5), 1 DEF ll2NewHistorySummary1, ll2NewHistorySummary2
The previous history summaries are equal to each other, because the fields of each record are equal.

The previous history summaries are equal to each other, because the fields of each record are equal. 

Because the fields are equal, the records are equal, but proving this requires that we prove the types of the records and invoke the HistorySummaryRecordCompositionLemma.

ideally, this qed step should just read:

Ideally, this QED step should just read:

However, the prover seems to get a little confused in this instance. We make life easier for the prover by defining some local variables and hiding their definitions before appealing to the HashCollisionResistant assumption.

(4.2) \(H2PreviousHistorySummary1.extension \in HashDomain\)

BY (2.1) DEF HistorySummaryType, HashDomain

(4.3) securedInput1 \in HashDomain

BY (3.4) DEF HashDomain

(4.4) \(H2PreviousHistorySummary2.extension \in HashDomain\)

BY (2.2) DEF HistorySummaryType, HashDomain

(4.5) securedInput2 \in HashDomain

BY (3.8) DEF HashDomain

(4.6) QED

Ideally, this QED step should just read:

BY (4.1), (4.2), (4.3), (4.4), (4.5), HashCollisionResistant

However, the prover seems to get a little confused in this instance. We make life easier for the prover by defining some local variables and hiding their definitions before appealing to the HashCollisionResistant assumption.

(5.1) \(h1a \triangleq H2PreviousHistorySummary1.extension\)

(5.2) \(h2a \triangleq securedInput1\)

(5.3) \(h1b \triangleq H2PreviousHistorySummary2.extension\)

(5.4) \(h2b \triangleq securedInput2\)

(5.5) \(h1a \in HashDomain\)

BY (4.2)

(5.6) \(h2a \in HashDomain\)

BY (4.3)

(5.7) \(h1b \in HashDomain\)

BY (4.4)

(5.8) \(h2b \in HashDomain\)

BY (4.5)

(5.9) \(Hash(h1a, h2a) = Hash(h1b, h2b)\)

BY (4.1) DEF newExtension1, newExtension2

(5.10) \(h1a = h1b \land h2a = h2b\)

(5.11) \(h1a, h2a, h1b, h2b\)

(5.12) QED

BY (5.1), (5.2), (5.3), (5.4), (5.5), HashCollisionResistant

(5.13) QED

BY (5.6)

The previous history summaries are equal to each other, because the fields of each record are equal.

Because the fields are equal, the records are equal, but proving this requires that we prove the types of the records and invoke the HistorySummaryRecordCompositionLemma.

(4.3) \(H2PreviousHistorySummary1 \in HistorySummaryType\)

BY (2.1)

(4.4) \(H2PreviousHistorySummary1 \in HistorySummaryType\)

BY (2.2)
The input values are equal to each other, because the secured inputs are equal, and the hash function is collision-resistant.

(3)16. \( \text{input1} = \text{input2} \)

(4)1. \( \text{securedInput1} = \text{securedInput2} \)

(4)2. \( \text{hashBarrier} \in \text{HashDomain} \)

(4)3. \( \text{input1} \in \text{HashDomain} \)

(4)4. \( \text{input2} \in \text{HashDomain} \)

(4)5. QED

By (4)1, (4)2, (4)3, (4)4, HistorySummaryRecordCompositionLemma

The Memoir-Basic previous history summaries are equal, because the Memoir-Opt previous history summaries are equal, and the matches are unique.

(2)4. \( \text{ll1PreviousHistorySummary1} = \text{ll1PreviousHistorySummary2} \)

(3)1. HistorySummariesMatch(\( \text{ll1PreviousHistorySummary1} \), \( \text{ll2PreviousHistorySummary1} \), \( \text{hashBarrier} \))

By (2)1

(3)2. HistorySummariesMatch(\( \text{ll1PreviousHistorySummary2} \), \( \text{ll2PreviousHistorySummary2} \), \( \text{hashBarrier} \))

By (2)2

(3)3. \( \text{ll2PreviousHistorySummary1} = \text{ll2PreviousHistorySummary2} \)

By (2)3

(3)4. QED

(4)1. \( \text{ll1PreviousHistorySummary1} \in \text{HashType} \)

By (2)1

(4)2. \( \text{ll1PreviousHistorySummary2} \in \text{HashType} \)

By (2)2

(4)3. \( \text{ll2PreviousHistorySummary1} \in \text{HistorySummaryType} \)

By (2)1

(4)4. \( \text{ll2PreviousHistorySummary2} \in \text{HistorySummaryType} \)

By (2)2

(4)5. \( \text{hashBarrier} \in \text{HashType} \)

By (1)1

(4)6. QED

By (3)1, (3)2, (3)3, (4)1, (4)2, (4)3, (4)4, (4)5, HistorySummariesMatchUniqueLemma

The Memoir-Basic history summaries are equal by construction, given that the inputs to the hash functions that produce these inputs are equal.

(2)5. QED

(3)1. \( \text{ll1HistorySummary1} = \text{Hash(\( \text{ll1PreviousHistorySummary1} \), \( \text{input1} \))} \)

By (2)1

(3)2. \( \text{ll1HistorySummary2} = \text{Hash(\( \text{ll1PreviousHistorySummary2} \), \( \text{input2} \))} \)

By (2)2

(3)3. \( \text{input1} = \text{input2} \)

By (2)3

(3)4. QED

By (2)4, (3)1, (3)2, (3)3
The base case and the recursive case are exhaustive.

(1)5. QED
   BY (1)3, (1)4
### 4.11 Proofs of Lemmas Relating to the Memoir-Opt Implementation

This module states and proves a bunch of lemmas that support the proof that the Memoir-Opt spec implements the Memoir-Basic spec.

This module includes the following theorems:
- `UnchangedAvailableInputsLemma`
- `UnchangedObservedOutputsLemma`
- `UnchangedObservedAuthenticatorsLemma`
- `UnchangedDiskPublicStateLemma`
- `UnchangedDiskPrivateStateEncLemma`
- `UnchangedDiskHistorySummaryLemma`
- `UnchangedDiskAuthenticatorLemma`
- `UnchangedDiskLemma`
- `UnchangedRAMPublicStateLemma`
- `UnchangedRAMPrivateStateEncLemma`
- `UnchangedRAMHistorySummaryLemma`
- `UnchangedRAMAuthenticatorLemma`
- `UnchangedRAMLemma`
- `UnchangedNVRAMHistorySummaryLemma`
- `UnchangedNVRAMSymmetricKeyLemma`
- `UnchangedNVRAMLemma`

#### EXTENDS MemoirLL2RefinementLemmas

The `UnchangedAvailableInputsLemma` states that when there is no change to the Memoir-Opt variable representing the available inputs, there is no change to the Memoir-Basic variable representing the available inputs.

**THEOREM** `UnchangedAvailableInputsLemma` $\Delta$

\[
\begin{align*}
\land \ \text{UNCHANGED } &\text{LL2AvailableInputs} \\
\land \ \text{LL2Refinement} \\
\land \ \text{LL2Refinement'} \\
\land \ \text{LL2TypeInvariant} \\
\land \ \text{LL2TypeInvariant'}
\end{align*}
\Rightarrow
\begin{align*}
\text{UNCHANGED } &\text{LL1AvailableInputs} \\
\langle 1\rangle 1. \text{ HAVE } &\land \ \text{UNCHANGED } \text{LL2AvailableInputs} \\
&\land \ \text{LL2Refinement} \\
&\land \ \text{LL2Refinement'} \\
&\land \ \text{LL2TypeInvariant} \\
&\land \ \text{LL2TypeInvariant'} \\
\langle 1\rangle 2. \text{ UNCHANGED LL2AvailableInputs} \\
\text{BY } &\langle 1\rangle 1 \\
\langle 1\rangle 3. \text{ LL1AvailableInputs } = \text{ LL2AvailableInputs} \\
\text{BY } &\langle 1\rangle 1 \text{ DEF LL2Refinement} \\
\langle 1\rangle 4. \text{ LL1AvailableInputs'} = \text{ LL2AvailableInputs'} \\
\text{BY } &\langle 1\rangle 1 \text{ DEF LL2Refinement} \\
\langle 1\rangle 5. \text{ QED} \\
\text{BY } &\langle 1\rangle 2, \langle 1\rangle 3, \langle 1\rangle 4
\end{align*}
\]

The `UnchangedObservedOutputsLemma` states that when there is no change to the Memoir-Opt variable representing the observed outputs, there is no change to the Memoir-Basic variable representing the observed outputs.
THEOREM UnchangedObservedOutputsLemma \(\triangleq\)

\[
\begin{align*}
& (\wedge \text{UNCHANGED LL2ObservedOutputs} \\
& \wedge \text{LL2Refinement} \\
& \wedge \text{LL2Refinement}' \\
& \wedge \text{LL2TypeInvariant} \\
& \wedge \text{LL2TypeInvariant}' ) \\
\Rightarrow \\
& \text{UNCHANGED LL1ObservedOutputs}
\end{align*}
\]

\(\langle 1 \rangle 1. \text{ HAVE } \wedge \text{UNCHANGED LL2ObservedOutputs} \\
\wedge \text{LL2Refinement} \\
\wedge \text{LL2Refinement}' \\
\wedge \text{LL2TypeInvariant} \\
\wedge \text{LL2TypeInvariant}' \)

\(\langle 1 \rangle 2. \text{UNCHANGED LL2ObservedOutputs} \)

BY \(\langle 1 \rangle 1\)

\(\langle 1 \rangle 3. \text{ LL1ObservedOutputs} = \text{LL2ObservedOutputs} \)

BY \(\langle 1 \rangle 1 \text{ DEF LL2Refinement}\)

\(\langle 1 \rangle 4. \text{ LL1ObservedOutputs}' = \text{LL2ObservedOutputs}' \)

BY \(\langle 1 \rangle 1 \text{ DEF LL2Refinement}\)

\(\langle 1 \rangle 5. \text{ QED} \)

BY \(\langle 1 \rangle 2, \langle 1 \rangle 3, \langle 1 \rangle 4\)

The UnchangedObservedAuthenticatorsLemma states that when there is no change to the Memoir-Opt variable representing the observed authenticators, there is no change to the Memoir-Basic variable representing the observed authenticators.
The UnchangedDiskPublicStateLemma states that when there is no change to the field of the Memoir-Opt variable representing the public state in the disk, there is no change to the field of the Memoir-Basic variable representing the public state in the disk.

Theorem UnchangedDiskPublicStateLemma

\[ (∀ \text{ UNCHANGED LL2Disk.publicState}) \Rightarrow (∀ \text{ UNCHANGED LL1Disk.publicState}) \]

1. Have \( ∀ \text{ UNCHANGED LL2Disk.publicState} \)
   - \( LL2Refinement \)
   - \( LL2Refinement' \)
   - \( LL2TypeInvariant \)
   - \( LL2TypeInvariant' \)
2. UNCHANGED LL2Disk.publicState
   by (1)1
3. LL1Disk.publicState = LL2Disk.publicState
   by (1)1 DEF LL2Refinement
4. LL1Disk.publicState' = LL2Disk.publicState'
   by (1)1 DEF LL2Refinement
5. QED
   by (1)2, (1)3, (1)4
The UnchangedDiskPrivateStateEncLemma states that when there is no change to the field of the Memoir-Opt variable representing the encrypted private state in the disk, there is no change to the field of the Memoir-Basic variable representing the encrypted private state in the disk.

\[
\text{THEOREM UnchangedDiskPrivateStateEncLemma} \triangleq \\
( \wedge \text{UNCHANGED LL2Disk.privateStateEnc} \\
\wedge \text{LL2Refinement} \\
\wedge \text{LL2Refinement'} \\
\wedge \text{LL2TypeInvariant} \\
\wedge \text{LL2TypeInvariant'}) \\
\Rightarrow \text{UNCHANGED LL1Disk.privateStateEnc} \\
\langle 1 \rangle 1. \text{HAVE} \wedge \text{UNCHANGED LL2Disk.privateStateEnc} \\
\wedge \text{LL2Refinement} \\
\wedge \text{LL2Refinement'} \\
\wedge \text{LL2TypeInvariant} \\
\wedge \text{LL2TypeInvariant'} \\
\langle 1 \rangle 2. \text{UNCHANGED LL2Disk.privateStateEnc} \\
\text{BY} \langle 1 \rangle 1 \\
\langle 1 \rangle 3. \text{LL1Disk.privateStateEnc} = \text{LL2Disk.privateStateEnc} \\
\text{BY} \langle 1 \rangle 1 \ \text{DEF} \ \text{LL2Refinement} \\
\langle 1 \rangle 4. \text{LL1Disk.privateStateEnc'} = \text{LL2Disk.privateStateEnc'} \\
\text{BY} \langle 1 \rangle 1 \ \text{DEF} \ \text{LL2Refinement} \\
\langle 1 \rangle 5. \text{QED} \\
\text{BY} \langle 1 \rangle 2, \langle 1 \rangle 3, \langle 1 \rangle 4 \\
\]

The UnchangedDiskHistorySummaryLemma states that when there is no change to the field of the Memoir-Opt variable representing the history summary in the disk, there is no change to the field of the Memoir-Basic variable representing the history summary in the disk.

\[
\text{THEOREM UnchangedDiskHistorySummaryLemma} \triangleq \\
( \wedge \text{UNCHANGED LL2Disk.historySummary} \\
\wedge \text{UNCHANGED LL2NVRAM.hashBarrier} \\
\wedge \text{LL2Refinement} \\
\wedge \text{LL2Refinement'} \\
\wedge \text{LL2TypeInvariant} \\
\wedge \text{LL2TypeInvariant'}) \\
\Rightarrow \text{UNCHANGED LL1Disk.historySummary} \\
\langle 1 \rangle 1. \text{HAVE} \wedge \text{UNCHANGED LL2Disk.historySummary} \\
\wedge \text{UNCHANGED LL2NVRAM.hashBarrier} \\
\wedge \text{LL2Refinement} \\
\wedge \text{LL2Refinement'} \\
\wedge \text{LL2TypeInvariant} \\
\wedge \text{LL2TypeInvariant'} \\
\langle 1 \rangle 2. \text{UNCHANGED LL2Disk.historySummary} \\
\text{BY} \langle 1 \rangle 1 \\
\langle 1 \rangle 3. \text{UNCHANGED LL2NVRAM.hashBarrier} \\
\text{BY} \langle 1 \rangle 1 \\
\langle 1 \rangle 4. \text{HistorySummariesMatch(} \\
\ \text{LL1Disk.historySummary,} \\
\text{LL2Disk.historySummary'\rangle
The UnchangedDiskAuthenticatorLemma states that when there is no change to the field of the Memoir-Opt variable representing the authenticator in the disk, there is no change to the field of the Memoir-Basic variable representing the authenticator in the disk.

**THEOREM UnchangedDiskAuthenticatorLemma**

\[ (\land \text{UNCHANGED } LL2_{\text{Disk}}.\text{authenticator} \land \text{UNCHANGED } LL2_{\text{NVRAM}}.\text{hashBarrier} \land LL2_{\text{Refinement}} \land LL2_{\text{Refinement'}} \land LL2_{\text{TypeInvariant}} \land LL2_{\text{TypeInvariant'}}) \]

\[ \Rightarrow \text{UNCHANGED } LL1_{\text{Disk}}.\text{authenticator} \]

\(\{1\}1\) \text{ HAVE } \land \text{UNCHANGED } LL2_{\text{Disk}}.\text{authenticator} \land \text{UNCHANGED } LL2_{\text{NVRAM}}.\text{hashBarrier} \land LL2_{\text{Refinement}} \land LL2_{\text{Refinement'}} \land LL2_{\text{TypeInvariant}} \land LL2_{\text{TypeInvariant'}}

\(\{1\}2\) \text{ UNCHANGED } LL2_{\text{Disk}}.\text{authenticator} \\
\text{BY } \{1\}1

\(\{1\}3\) \text{ UNCHANGED } LL2_{\text{NVRAM}}.\text{hashBarrier} \\
\text{BY } \{1\}1

\(\{1\}4\) \exists \text{symmetricKey } \in \text{SymmetricKeyType} : \text{AuthenticatorsMatch(} \\
\text{LL1_{Disk}.authenticator,} \\
\text{LL2_{Disk}.authenticator,} \\
\text{symmetricKey,} \\
\text{LL2_{NVRAM}.hashBarrier})

\text{BY } \{1\}1 \text{ DEF } LL2_{\text{Refinement}}
The UnchangedDiskLemma states that when there is no change to the Memoir-Opt variable representing the disk, there is no change to the Memoir-Basic variable representing the disk.

**Theorem UnchangedDiskLemma**

\[
\Delta \equiv \\
(\land \text{UNCHANGED } LL2Disk \\
\land \text{UNCHANGED } LL2NVRAM.symmetricKey \\
\land \text{UNCHANGED } LL2NVRAM.hashBarrier \\
\land \text{LL2Refinement} \\
\land \text{LL2Refinement'} \\
\land \text{LL2TypeInvariant} \\
\land \text{LL2TypeInvariant'}) \\
\Rightarrow \\
\text{UNCHANGED } LL1Disk
\]

1. **Case 1**

   \[\land \text{UNCHANGED } LL2Disk \\
   \land \text{UNCHANGED } LL2NVRAM.symmetricKey \\
   \land \text{UNCHANGED } LL2NVRAM.hashBarrier \\
   \land \text{LL2Refinement} \\
   \land \text{LL2Refinement'} \\
   \land \text{LL2TypeInvariant} \\
   \land \text{LL2TypeInvariant'}\]

   **By (1)**

2. **Case 2**

   \[\land \text{UNCHANGED } LL1Disk.publicState \\
   \text{By (2)1} \]

   **By (1)**

3. **Case 3**

   \[\land \text{UNCHANGED } LL1Disk.privateStateEnc \\
   \text{By (2)1} \]

   **By (1)**

4. **Case 4**

   \[\land \text{UNCHANGED } LL1Disk.historySummary \\
   \text{By (2)2} \]
The UnchangedRAMPublicStateLemma states that when there is no change to the field of the Memoir-Opt variable representing the public state in the RAM, there is no change to the field of the Memoir-Basic variable representing the public state in the RAM.

\[ \text{Theorem } \text{UnchangedRAMPublicStateLemma} \triangleq \\
(\land \text{UNCHANGED } LL2\text{RAM.publicState} \\
\land LL2\text{Refinement} \\
\land LL2\text{Refinement}' \\
\land LL2\text{TypeInvariant} \\
\land LL2\text{TypeInvariant}') \\
\Rightarrow \\
\text{UNCHANGED } LL1\text{RAM.publicState} \]

\[ \begin{aligned}
\langle 1 \rangle 1. & \text{ HAVE } \land \text{UNCHANGED } LL2\text{RAM.publicState} \\
& \land LL2\text{Refinement} \\
& \land LL2\text{Refinement}' \\
& \land LL2\text{TypeInvariant} \\
& \land LL2\text{TypeInvariant}' \\
\langle 1 \rangle 2. & \text{ UNCHANGED } LL2\text{RAM.publicState} \\
\langle 1 \rangle 1 \\
\langle 1 \rangle 3. & \text{ LL1RAM.publicState} = LL2\text{RAM.publicState} \\
\langle 1 \rangle 1 \\
\langle 1 \rangle 4. & \text{ LL1RAM.publicState}' = LL2\text{RAM.publicState}' \\
\langle 1 \rangle 1 \\
\langle 1 \rangle 5. & \text{ QED} \\
\langle 1 \rangle 2, \langle 1 \rangle 3, \langle 1 \rangle 4
\end{aligned} \]

The UnchangedRAMPrivateStateEncLemma states that when there is no change to the field of the Memoir-Opt variable representing the encrypted private state in the RAM, there is no change to the field of the Memoir-Basic variable representing the encrypted private state in the RAM.

\[ \text{Theorem } \text{UnchangedRAMPrivateStateEncLemma} \triangleq \\
(\land \text{UNCHANGED } LL2\text{RAM.privateStateEnc} \\
\land LL2\text{Refinement} \\
\land LL2\text{Refinement}') \]

\[ \begin{aligned}
\langle 1 \rangle 1. & \text{ HAVE } \land \text{UNCHANGED } LL2\text{RAM.privateStateEnc} \\
& \land LL2\text{Refinement} \\
& \land LL2\text{Refinement}' \\
\langle 1 \rangle 2, \langle 1 \rangle 3, \langle 1 \rangle 4
\end{aligned} \]
The UnchangedRAMHistorySummaryLemma states that when there is no change to the field of the Memoir-Opt variable representing the history summary in the RAM, there is no change to the field of the Memoir-Basic variable representing the history summary in the RAM.
The UnchangedRAMAuthenticatorLemma states that when there is no change to the field of the Memoir-Opt variable representing the authenticator in the RAM, there is no change to the field of the Memoir-Basic variable representing the authenticator in the RAM.

**THEOREM** UnchangedRAMAuthenticatorLemma \(\Delta\)  
\(\land\) UNCHANGED LL2RAM.authenticator  
\(\land\) LL2Refinement  
\(\land\) LL2NVRA.M.hashBarrier  
\(\land\) LL2Refinement'  
\(\land\) LL2TypeInvariant  
\(\land\) LL2TypeInvariant')  
\(\Rightarrow\)  
UNCHANGED LL1RAM.authenticator  

\(\langle 1\rangle.1.\) HAVE  
\(\land\) UNCHANGED LL2RAM.authenticator  
\(\land\) UNCHANGED LL2NVRA.M.hashBarrier  
\(\land\) LL2Refinement  
\(\land\) LL2Refinement'  
\(\land\) LL2TypeInvariant  
\(\land\) LL2TypeInvariant')  

\(\langle 1\rangle.2.\) UNCHANGED LL2RAM.authenticator  
BY \(\langle 1\rangle.1\)  

\(\langle 1\rangle.3.\) UNCHANGED LL2NVRA.M.hashBarrier  
BY \(\langle 1\rangle.1\)  

\(\langle 1\rangle.4.\) \(\exists\) symmetricKey \(\in\) SymmetricKeyType :  
AuthenticatorsMatch(  
LL1RAM.authenticator,  
LL2RAM.authenticator,  
symmetricKey,  
LL2NVRA.M.hashBarrier)  
BY \(\langle 1\rangle.1\) DEF LL2Refinement  

\(\langle 1\rangle.5.\) \(\exists\) symmetricKey \(\in\) SymmetricKeyType :  
AuthenticatorsMatch(  
LL1RAM.authenticator',  
LL2RAM.authenticator',  
symmetricKey,  
LL2NVRA.M.hashBarrier')  
BY \(\langle 1\rangle.1\) DEF LL2Refinement  

\(\langle 1\rangle.6.\) QED
The *UnchangedRAMLemma* states that when there is no change to the Memoir-Opt variable representing the RAM, there is no change to the Memoir-Basic variable representing the RAM.

**THEOREM** *UnchangedRAMLemma* $\triangleq$

\[
(\land \text{UNCHANGED } L\!L_2\!R\!A\!M)
\land \text{UNCHANGED } L\!L_2\!N\!V\!R\!A\!M.\!\text{symmetricKey}
\land \text{UNCHANGED } L\!L_2\!N\!V\!R\!A\!M.\!\text{hashBarrier}
\land L\!L_2\!\text{Refinement}
\land L\!L_2\!\text{Refinement}'
\land L\!L_2\!\text{TypeInvariant}
\land L\!L_2\!\text{TypeInvariant}'
\Rightarrow
\text{UNCHANGED } L\!L_1\!R\!A\!M
\]

(1)1. HAVE $\land$ UNCHANGED *LL2RAM*

\[
\land \text{UNCHANGED } L\!L_2\!N\!V\!R\!A\!M.\!\text{symmetricKey}
\land \text{UNCHANGED } L\!L_2\!N\!V\!R\!A\!M.\!\text{hashBarrier}
\land L\!L_2\!\text{Refinement}
\land L\!L_2\!\text{Refinement}'
\land L\!L_2\!\text{TypeInvariant}
\land L\!L_2\!\text{TypeInvariant}'
\]

(1)2. UNCHANGED *LL1RAM_.publicState*

(2)1. UNCHANGED *LL2RAM_.publicState*

(2)2. QED

(1)3. UNCHANGED *LL1RAM_.privateStateEnc*

(2)1. UNCHANGED *LL2RAM_.privateStateEnc*

(2)2. QED

(1)4. UNCHANGED *LL1RAM_.historySummary*

(2)1. UNCHANGED *LL2RAM_.historySummary*

(2)2. QED

(1)5. UNCHANGED *LL1RAM_.authenticator*

(2)1. UNCHANGED *LL2RAM_.authenticator*

(2)2. QED
The UnchangedNVRAMHistorySummaryLemma states that when there is no change to the $LL_2$NVRAMLogicalHistorySummary value, which represents the logical value of the history summary in the NVRAM and SPCR of the Memoir-Opt spec, there is no change to the field of the Memoir-Basic variable representing the history summary in the NVRAM.

\begin{align*}
\text{THEOREM UnchangedNVRAMHistorySummaryLemma } \triangleq & \\
( & \land \text{UNCHANGED } LL_2\text{NVRAMLogicalHistorySummary} \\
& \land \text{UNCHANGED } LL_2\text{NVRAM.symmetricKey} \\
& \land \text{UNCHANGED } LL_2\text{NVRAM.hashBarrier} \\
& \land LL_2\text{Refinement} \\
& \land LL_2\text{Refinement'} \\
& \land LL_2\text{TypeInvariant} \\
& \land LL_2\text{TypeInvariant'}) \\
\Rightarrow & \\
\text{UNCHANGED } LL_1\text{NVRAM.historySummary} \\
\end{align*}

\begin{align*}
\langle 1 \rangle & \text{. HAVE } ( & \land \text{UNCHANGED } LL_2\text{NVRAMLogicalHistorySummary} \\
& \land \text{UNCHANGED } LL_2\text{NVRAM.symmetricKey} \\
& \land \text{UNCHANGED } LL_2\text{NVRAM.hashBarrier} \\
& \land LL_2\text{Refinement} \\
& \land LL_2\text{Refinement'} \\
& \land LL_2\text{TypeInvariant} \\
& \land LL_2\text{TypeInvariant'}) \\
\langle 1 \rangle & \text{. UNCHANGED } LL_2\text{NVRAMLogicalHistorySummary} \\
\langle 2 \rangle & \text{. UNCHANGED } LL_2\text{NVRAM.symmetricKey} \\
\langle 3 \rangle & \text{. UNCHANGED } LL_2\text{NVRAM.hashBarrier} \\
\langle 5 \rangle & \text{. HistorySummariesMatch}( \\
& LL_1\text{NVRAM.historySummary}, \\
& LL_2\text{NVRAMLogicalHistorySummary}, \\
& LL_2\text{NVRAM.hashBarrier}) \\
\langle 6 \rangle & \text{. HistorySummariesMatch}( \\
& LL_1\text{NVRAM.historySummary'}, \\
& LL_2\text{NVRAMLogicalHistorySummary'}, \\
& LL_2\text{NVRAM.hashBarrier'}) \\
\langle 7 \rangle & \text{. QED} \\
\langle 2 \rangle & LL_1\text{NVRAM.historySummary} \in \text{HashType} \\
\langle 2 \rangle & LL_1\text{NVRAM.historySummary'} \in \text{HashType} \\
\end{align*}
(2)3. \( LL2NVRAML\text{LogicalHistorySummary} \in HistorySummaryType \)
   BY (1)1, \( LL2NVRAML\text{LogicalHistorySummaryTypeSafe} \)

(2)4. \( LL2NVRAM.\text{hashBarrier} \in HashType \)
   BY (1)1, \( LL2SubtypeImplicationLemma\) def \( LL2SubtypeImplication \)

(2)5. QED
   BY (1)2, (1)3, (1)4, (1)5, (1)6, (2)1, (2)2, (2)3, (2)4,
   HistorySummariesMatchUniqueLemma

The \( UnchangedNVRASS\)ymmetricKeyLemma states that when there is no change to the field of the Memoir-Opt variable representing the symmetric key in the RAM, there is no change to the field of the Memoir-Basic variable representing the symmetric key in the RAM.

**THEOREM** \( UnchangedNVRASS\)ymmetricKeyLemma \( \triangleq \)

\[
\begin{align*}
& (\wedge \text{UNCHANGED } LL2NVRAM.\text{symmetricKey} \\
& \wedge LL2Refinement \\
& \wedge LL2Refinement' \\
& \wedge LL2TypeInvariant \\
& \wedge LL2TypeInvariant') \\
\Rightarrow \\
& \text{UNCHANGED } LL1NVRAM.\text{symmetricKey} \\
\end{align*}
\]

\( 1)1. \) HAVE \( \wedge \text{UNCHANGED } LL2NVRAM.\text{symmetricKey} \\
\wedge LL2Refinement \\
\wedge LL2Refinement' \\
\wedge LL2TypeInvariant \\
\wedge LL2TypeInvariant') \\

\( 1)2. \) UNCHANGED \( LL2NVRAM.\text{symmetricKey} \\
BY (1)1 \\

\( 1)3. \) \( LL1NVRAM.\text{symmetricKey} = LL2NVRAM.\text{symmetricKey} \\
BY (1)1 def LL2Refinement \\

\( 1)4. \) \( LL1NVRAM.\text{symmetricKey}' = LL2NVRAM.\text{symmetricKey}' \\
BY (1)1 def LL2Refinement \\

\( 1)5. \) QED \\
BY (1)2, (1)3, (1)4

The \( UnchangedNVRA\)Lemma states that when there is no change to the Memoir-Opt variables representing the \( NVRAM \) and \( SPCR \), there is no change to the Memoir-Basic variable representing the \( NVRAM \).

**THEOREM** \( UnchangedNVRA\) Lemma \( \triangleq \)

\[
\begin{align*}
& (\wedge \text{UNCHANGED } LL2NVRAM \\
& \wedge \text{UNCHANGED } LL2SPCR \\
& \wedge LL2Refinement \\
& \wedge LL2Refinement' \\
& \wedge LL2TypeInvariant \\
& \wedge LL2TypeInvariant') \\
\Rightarrow \\
& \text{UNCHANGED } LL1NVRAM \\
\end{align*}
\]

\( 1)1. \) HAVE \( \wedge \text{UNCHANGED } LL2NVRAM \\
\wedge \text{UNCHANGED } LL2SPCR \\
\wedge LL2Refinement \\
\wedge LL2Refinement' \\

282
\[ LL_2 \text{TypeInvariant} \]
\[ LL_2 \text{TypeInvariant}' \]

1. UNCHANGED \( LL_1\text{NVRAM.historySummary} \)
2. BY (1) \(\text{DEF } LL_2\text{NVRAMLogicalHistorySummary} \)
3. QED
4. BY (1), (2), \(\text{UnchangedNVRAMHistorySummaryLemma} \)
5. UNCHANGED \( LL_1\text{NVRAM.symmetricKey} \)
6. BY (1) \(\text{DEF } LL_2\text{NVRAM.symmetricKey} \)
7. QED
8. BY (1), (2), \(\text{UnchangedNVRAMSymmetricKeyLemma} \)
9. \( LL_1\text{NVRAM} \in LL_1\text{TrustedStorageType} \)
10. BY (1) \(\text{DEF } LL_2\text{Refinement} \)
11. \( LL_1\text{NVRAM}' \in LL_1\text{TrustedStorageType} \)
12. BY (1) \(\text{DEF } LL_2\text{Refinement} \)
13. QED
14. BY (1), (2), (3), (4), (5), \(\text{LL1NVRAMRecordCompositionLemma} \)
This module proves that the Memoir-Opt spec implements the Memoir-Basic spec, under the defined refinement.

**EXTENDS** MemoirLL2Implementation

The *LL2Implementation* theorem is where the rubber meets the road. This is the ultimate proof that the Memoir-Opt spec implements the Memoir-Basic spec, under the defined refinement.

**THEOREM** LL2Implementation $\triangleq$ LL2Spec $\land \Box LL2Refinement \Rightarrow LL1Spec

This proof will require the LL2TypeInvariant. Fortunately, the LL2TypeSafe theorem has already proven that the Memoir-Opt spec satisfies its type invariant.

1. **LL2Spec $\Rightarrow \Box LL2TypeInvariant**
   
   **BY** LL2TypeSafe

   The top level of the proof is boilerplate TLA+ for a StepSimulation proof. First, we prove that the initial predicate of the Memoir-Opt spec, conjoined with the LL2Refinement and type invariant, implies the initial predicate of the Memoir-Basic spec. Second, we prove that the LL2Next predicate, conjoined with the LL2Refinement and type invariant in both primed and unprimed states, implies the LL1Next predicate. Third, we use temporal induction to prove that these two conditions imply that, if the LL2Refinement and the type invariant always hold, the LL2Spec implies the LL1Spec.

2. **LL2Init $\land LL2Refinement \land LL2TypeInvariant \Rightarrow LL1Init**
   
   **We begin the base case by assuming the antecedent.**

   **We pick a symmetric key and a hash barrier that satisfy the LL2Init predicate.**

   **We re-state the definitions from LL2Init.**

   2. **initialPrivateStateEnc $\triangleq$ SymmetricEncrypt(symmetricKey, InitialPrivateState)**
   2. **initialStateHash $\triangleq$ Hash(InitialPublicState, initialPrivateStateEnc)**
   2. **ll2InitialHistorySummary $\triangleq$ [**anchor** $\mapsto$ BaseHashValue, **extension** $\mapsto$ BaseHashValue]**
   2. **ll2InitialHistorySummaryHash $\triangleq$ Hash(BaseHashValue, BaseHashValue)**
   2. **ll2InitialHistoryStateBinding $\triangleq$ Hash(ll2InitialHistorySummaryHash, initailStateHash)**
   2. **ll2InitialAuthenticator $\triangleq$ GenerateMAC(symmetricKey, ll2InitialHistoryStateBinding)**
   2. **ll2InitialUntrustedStorage $\triangleq$ [**publicState** $\mapsto$ InitialPublicState, **privateStateEnc** $\mapsto$ initialPrivateStateEnc, **historySummary** $\mapsto$ ll2InitialHistorySummary, **authenticator** $\mapsto$ ll2InitialAuthenticator]**
   2. **ll2InitialTrustedStorage $\triangleq$ [**historySummaryAnchor** $\mapsto$ BaseHashValue, **symmetricKey** $\mapsto$ symmetricKey, **hashBarrier** $\mapsto$ hashBarrier, **extensionInProgress** $\mapsto$ FALSE]**

   **We prove that the definitions from LL2Init satisfy their types, using the LL2InitDefsTypeSafeLemma.**

   2. **$\&$ initialPrivateStateEnc $\in$ PrivateStateEncType
   $\&$ initialStateHash $\in$ HashType
   $\&$ ll2InitialHistorySummary $\in$ HistorySummaryType
   $\&$ ll2InitialHistorySummaryHash $\in$ HashType
   $\&$ ll2InitialHistoryStateBinding $\in$ HashType
match across the two specs. We will use this in three places below: for LL1Disk.\textit{authenticator}, LL1RAM.\textit{authenticator}, and LL1Observed\textit{Authenticators}. We prove the match using the definition of the \textit{AuthenticatorsMatch} predicate.

(2)6. \textit{AuthenticatorsMatch}(ll1Initial\textit{Authenticator},

\newpage

\begin{align}
\wedge & \text{ll2Initial\textit{Authenticator} }\in\text{ MACType} \\
\wedge & \text{ll2Initial\textit{UntrustedStorage} }\in\text{ LL2\textit{UntrustedStorageType}} \\
\wedge & \text{ll2Initial\textit{TrustedStorage} }\in\text{ LL2\textit{TrustedStorageType}} \\
(3)2.\text{ QED} & \\
& \text{by (3)1, }\text{LL2InitDefsTypeSafeLemma}
\end{align}
First, we prove some types needed by the definition of the AuthenticatorsMatch predicate.

1. **initialStateHash ∈ HashType**
   - By (2)

2. **BaseHashValue ∈ HashType**
   - By ConstantsTypeSafe

3. **ll2InitialHistorySummary ∈ HistorySummaryType**
   - By (2)

We then prove that, in the Memoir-Opt spec, the initial authenticator is a valid MAC for the initial history state binding. We will use the MACComplete property.

4. **ValidateMAC( LL2NVRAM.symmetricKey, ll2InitialHistoryStateBinding, ll2InitialAuthenticator )**

In the Memoir-Opt spec, the initial authenticator is generated as a MAC of the initial history state binding.

5. **ll2InitialAuthenticator = GenerateMAC( LL2NVRAM.symmetricKey, ll2InitialHistoryStateBinding )**

We can thus use the MACComplete property to show that the generated MAC validates appropriately. To do this, we first need to prove some types.

6. **LL2NVRAM.symmetricKey ∈ SymmetricKeyType**
   - By (2)

7. **ll2InitialHistoryStateBinding ∈ HashType**
   - By (2)

Then, we appeal to the MACComplete property in a straightforward way.

8. **QED**

We then prove that, in the Memoir-Basic spec, the initial authenticator is generated as a MAC of the initial history state binding.

9. **ll1InitialAuthenticator = GenerateMAC( LL2NVRAM.symmetricKey, ll1InitialHistoryStateBinding )**

The initial history summaries match across the two specs, as we proved above.

10. **HistorySummariesMatch( BaseHashValue, ll2InitialHistorySummary, LL2NVRAM.hashBarrier )**
    - By (2)

We then invoke the definition of the AuthenticatorsMatch predicate.

11. **QED**
    - By (3), (3), (3), (3), (3), (3), (3), (3), (3)
def AuthenticatorsMatch, ll2InitialAuthenticator,
     ll1InitialHistoryStateBinding, ll2InitialHistoryStateBinding,
     ll2InitialHistorySummaryHash, ll2InitialHistorySummary

The next six steps assert each conjunct in the \( LL1Init \) predicate. For the \( LL1Disk \) variable, we prove each field within the record separately.

\[ 2 \]\( 7. \) \( LL1Disk = ll1InitialUntrustedStorage \)

The \( LL1Disk \)'s public state equals the initial public state, because the refinement asserts a direct equality between this field for the two specs.

\[ 3 \]\( 1. \) \( LL1Disk.publicState = InitialPublicState \)

\[ 4 \]\( 1. \) \( LL2Disk.publicState = InitialPublicState \)

\[ 4 \]\( 2. \) \( LL1Disk.publicState = LL2Disk.publicState \)

\[ 4 \]\( 3. \) QED

The \( LL1Disk \)'s private encrypted state equals the initial private encrypted state, because the refinement asserts a direct equality between this field for the two specs.

\[ 3 \]\( 2. \) \( LL1Disk.privateStateEnc = initialPrivateStateEnc \)

\[ 4 \]\( 1. \) \( LL2Disk.privateStateEnc = initialPrivateStateEnc \)

\[ 4 \]\( 2. \) \( LL1Disk.privateStateEnc = LL2Disk.privateStateEnc \)

\[ 4 \]\( 3. \) QED

The \( LL1Disk \)'s history summary equals the base hash value. There are four steps: (1) The refinement asserts that the corresponding fields from the two specs match according to the \( HistorySummariesMatch \) predicate. (2) We prove that the initial history summary values match across the two specs. (3) We prove that the Memoir-Opt disk’s history summary equals its initial history summary value. (4) We use the \( HistorySummariesMatchUniqueLemma \) to prove that the field in the Memoir-Basic spec must equal the base hash value.

\[ 3 \]\( 3. \) \( LL1Disk.historySummary = BaseHashValue \)

The corresponding fields from the two specs match, thanks to the refinement.

\[ 4 \]\( 1. \) \( HistorySummariesMatch( \)

\[ 4 \]\( 2. \) \( LL1Disk.historySummary, \)

\[ 4 \]\( 3. \) \( LL2Disk.historySummary, \)

\[ 4 \]\( 4. \) \( LL2NVRAM.hashBarrier) \)

\[ 4 \]\( 1. \) QED

The history summary in the Memoir-Opt spec’s disk equals the initial history summary, by the definition of \( LL2Init \).

\[ 4 \]\( 3. \) \( LL2Disk.historySummary = ll2InitialHistorySummary \)

\[ 4 \]\( 4. \) QED

We use the \( HistorySummariesMatchUniqueLemma \) to prove that the field in the Memoir-Basic spec equals the base hash value. This requires proving some types.

\[ 5 \]\( 1. \) \( LL1Disk.historySummary \in HashType \)

\[ 5 \]\( 2. \) \( BaseHashValue \in HashType \)

\[ 5 \]\( 3. \) \( LL2Disk.historySummary \in HistorySummaryType \)

287
The LL1Disk's authenticator equals the initial authenticator. There are four steps: (1) The refinement asserts that the corresponding fields from the two specs match according to the HistorySummariesMatch predicate. (2) We prove that the initial authenticator values match across the two specs. (3) We prove that the Memoir-Opt disk's authenticator equals its initial authenticator value. (4) We use the AuthenticatorsMatchUniqueLemma to prove that the field in the Memoir-Basic spec must equal the initial authenticator value.

(3.4) LL1Disk.authenticator = ll1InitialAuthenticator

The corresponding fields from the two specs match, thanks to the refinement.

(4.1) ∃ symmetricKey1 ∈ SymmetricKeyType :
    AuthenticatorsMatch(
    LL1Disk.authenticator,
    LL2Disk.authenticator,
    symmetricKey1,
    LL2NVRAM.hashBarrier)

BY (2.1) DEF LL2Refinement

The corresponding initial authenticator values match across the two specs. We proved this above.

(4.2) AuthenticatorsMatch(
    ll1InitialAuthenticator,
    ll2InitialAuthenticator,
    LL2NVRAM.symmetricKey,
    LL2NVRAM.hashBarrier)

BY (2.6)

The Memoir-Opt disk's authenticator equals its initial authenticator value, as asserted by the LL2Init predicate.

(4.3) LL2Disk.authenticator = ll2InitialAuthenticator

BY (2.2) DEF ll2InitialAuthenticator, ll2InitialHistoryStateBinding,

ll2InitialHistorySummaryHash, initialStateHash, initialPrivateStateEnc

We use the AuthenticatorsMatchUniqueLemma to prove that the field in the Memoir-Basic spec equals the initial authenticator value. This requires proving some types.

(4.4) QED

(5.1) LL1Disk.authenticator ∈ MACType

BY (2.1) DEF LL2Refinement, LL1UntrustedStorageType

(5.2) ll1InitialAuthenticator ∈ MACType

BY (2.4)

(5.3) LL2Disk.authenticator ∈ MACType

BY (2.1), LL2SubtypeImplication Lemma DEF LL2SubtypeImplication

(5.4) LL2NVRAM.symmetricKey ∈ SymmetricKeyType

BY (2.1), LL2SubtypeImplication Lemma DEF LL2SubtypeImplication

(5.5) LL2NVRAM.hashBarrier ∈ HashType

BY (2.1), LL2SubtypeImplication Lemma DEF LL2SubtypeImplication

(5.6) QED

BY (4.1), (4.2), (4.3), (5.1), (5.5), (5.3), (5.4), (5.5), AuthenticatorsMatchUniqueLemma

The refinement asserts that the Disk record has the appropriate type.

(3.5) LL1Disk ∈ LL1UntrustedStorageType

BY (2.1) DEF LL2Refinement

We use the LL1DiskRecordCompositionLemma to unify the field equalities into a record equality.
We use the $\text{HistorySummariesMatchUniqueLemma}$ to prove that the field in the Memoir-Basic spec must equal the base hash value. This requires proving some types.

(3.5) BaseHashValue $\in$ HashType
  by (2.1) $\text{LL2SubtypeImplicationLemma}$, $\text{LL1UntrustedStorageType}$

The initial history summary matches across the two specs, as we proved above.

(4.3) $\text{LL1RAM} . \text{historySummary} = \text{ll2InitialHistorySummary}$
  by (2.2) $\text{ll2InitialHistorySummary}$

We use the $\text{HistorySummariesMatchUniqueLemma}$ to prove that the field in the Memoir-Basic spec must equal the base hash value.

(4.4) QED

The history summary in the Memoir-Opt spec’s RAM equals the initial history summary, by the definition of $\text{LL1Init}$.
prove that the field in the Memoir-Basic spec must equal the initial authenticator value.

AuthenticatorsMatchUniqueLemma

RAM's authenticator equals its initial authenticator value. (4)

We use the

We prove that the initial authenticator values match across the two specs. (3)

We prove that the Memoir-Opt RAM's authenticator equals its initial authenticator value. (4) We use the AuthenticatorsMatchUniqueLemma to prove that the field in the Memoir-Opt spec must equal the initial authenticator value.

(3.4). \( \text{LL1RAM.authenticator} = \text{ll1InitialAuthenticator} \)

The corresponding fields from the two specs match, thanks to the refinement.

(4.1). \( \exists \text{symmetricKey1} \in \text{SymmetricKeyType} : \)

\[
\text{AuthenticatorsMatch}(
\text{LL1RAM.authenticator},
\text{LL2RAM.authenticator},
\text{symmetricKey1},
\text{LL2NVRAM.hashBarrier})
\]

By (2.1) DEF LL2Refinement

The corresponding initial authenticator values match across the two specs. We proved this above.

(4.2). \( \text{AuthenticatorsMatch}(
\text{ll1InitialAuthenticator},
\text{ll2InitialAuthenticator},
\text{LL2NVRAM.symmetricKey},
\text{LL2NVRAM.hashBarrier}) \)

By (2.6)

The Memoir-Opt RAM's authenticator equals its initial authenticator value, as asserted by the LL2Init predicate.

(4.3). \( \text{LL2RAM.authenticator} = \text{ll2InitialAuthenticator} \)

By (2.2)

DEF \( \text{ll2InitialAuthenticator}, \text{ll2InitialHistoryStateBinding}, \)

\( \text{ll2InitialHistorySummaryHash}, \text{initialStateHash}, \text{initialPrivateKeyEnc} \)

We use the AuthenticatorsMatchUniqueLemma to prove that the field in the Memoir-Opt spec equals the initial authenticator value. This requires proving some types.

(4.4). QED

(5.1). \( \text{LL1RAM.authenticator} \in \text{MECType} \)

By (2.1) DEF LL2Refinement, LL1UntrustedStorageType

(5.2). \( \text{ll1InitialAuthenticator} \in \text{MECType} \)

By (2.4)

(5.3). \( \text{LL2RAM.authenticator} \in \text{MECType} \)

By (2.1), LL2SubtypeImplicationLemma DEF LL2SubtypeImplication

(5.4). \( \text{LL2NVRAM.symmetricKey} \in \text{SymmetricKeyType} \)

By (2.1), LL2SubtypeImplicationLemma DEF LL2SubtypeImplication

(5.5). \( \text{LL2NVRAM.hashBarrier} \in \text{HashType} \)

By (2.1), LL2SubtypeImplicationLemma DEF LL2SubtypeImplication

(5.6). QED

By (4.1), (4.2), (4.3), (5.1), (5.2), (5.3), (5.4), (5.5), AuthenticatorsMatchUniqueLemma

The refinement asserts that the RAM record has the appropriate type.

(3.5). \( \text{LL1RAM} \in \text{LL1UntrustedStorageType} \)

By (2.1) DEF LL2Refinement

We use the LL1RAMRecordCompositionLemma to unify the field equalities into a record equality.

(3.6). QED

By (3.1), (3.2), (3.3), (3.4), (3.5), LL1RAMRecordCompositionLemma
DEF ll1InitialUntrustedStorage

The third conjunct in the LL1Init predicate relates to the LL1NVRAM variable.

(2)9. LL1NVRAM = ll1InitialTrustedStorage

The LL1NVRAM's history summary equals the base hash value. There are three steps: (1) The refinement asserts that the corresponding fields from the two specs match according to the HistorySummariesMatch predicate. (2) We prove that the initial history summary values match across the two specs. (3) We prove that the Memoir-Opt NVRAM's logical history summary equals its initial history summary value. (4) We use the HistorySummariesMatchUniqueLemma to prove that the field in the Memoir-Basic spec must equal the base hash value.

(3)1. LL1NVRAM.historySummary = BaseHashValue

The corresponding fields from the two specs match, thanks to the refinement.

(4)1. HistorySummariesMatch(
    LL1NVRAM.historySummary, LL2NVRAMLlogicalHistorySummary, LL2NVRAM.hashBarrier)
   BY (2)1 DEF LL2Refinement

The initial history summaries match across the two specs, as we proved above.

(4)2. HistorySummariesMatch(BaseHashValue, ll2InitialHistorySummary, LL2NVRAM.hashBarrier)
   BY (2)5

The logical value of the history summary in the Memoir-Opt spec's NVRAM equals the initial history summary. This follows from the definition of LL2NVRAMLlogicalHistorySummary.

(4)3. LL2NVRAMLlogicalHistorySummary = ll2InitialHistorySummary
   (5)1. LL2NVRAM.extensionInProgress = FALSE
      BY (2)2
   (5)2. LL2NVRAM.historySummaryAnchor = BaseHashValue
      BY (2)2
   (5)3. LL2SPCR = BaseHashValue
      BY (2)2
   (5)4. QED
      BY (5)1, (5)2, (5)3 DEF LL2NVRAMLlogicalHistorySummary, ll2InitialHistorySummary

We use the HistorySummariesMatchUniqueLemma to prove that the field in the Memoir-Basic spec equals the base hash value. This proves some types.

(4)4. QED
   (5)1. LL1NVRAM.historySummary \in HashType
      BY (2)1 DEF LL2Refinement, LL1TrustedStorageType
   (5)2. BaseHashValue \in HashType
      BY ConstantsTypeSafe
   (5)3. LL2NVRAMLlogicalHistorySummary \in HistorySummaryType
      BY (2)1, LL2NVRAMLlogicalHistorySummaryTypeSafe
   (5)4. LL2NVRAM.hashBarrier \in HashType
      BY (2)1, LL2SubtypeImplicationLemma\[DEF LL2SubtypeImplication
   (5)5. QED
      BY (4)1, (4)2, (4)3, (5)1, (5)2, (5)3, (5)4, HistorySummariesMatchUniqueLemma

The symmetric key in the LL1NVRAM matches its initial value by direct equality through the refinement.

(3)2. LL1NVRAM.symmetricKey = symmetricKey
   (4)1. LL1NVRAM.symmetricKey = LL2NVRAM.symmetricKey
      BY (2)1 DEF LL2Refinement
   (4)2. LL2NVRAM.symmetricKey = symmetricKey
      BY (2)2
   (4)3. QED
      BY (4)1, (4)2

The refinement asserts that the NVRAM record has the appropriate type.

(3)3. LL1NVRAM \in LL1TrustedStorageType
We use the \textit{LL1NVRAMRecordCompositionLemma} to unify the field equalities into a record equality.

\\[\text{(3)4. QED}\\]

\textbf{By (3)1, (3)2, (3)3, LL1NVRAMRecordCompositionLemma}

\textbf{Def} \textit{ll1InitialTrustedStorage}\\

The fourth conjunct in the \textit{LL1Init} predicate relates to the \textit{LL1AvailableInputs} variable. The proof is straightforward, because the equality is direct.

\\[\text{(2)10. LL1AvailableInputs} = \text{InitialAvailableInputs}\\]

\textbf{(3)1. LL2AvailableInputs} = \text{InitialAvailableInputs}\\

\textbf{By (2)2}\\

\textbf{(3)2. LL1AvailableInputs} = \text{LL2AvailableInputs}\\

\textbf{By (2)1 Def LL2Refinement}\\

\textbf{(3)3. QED}\\

\textbf{By (3)1, (3)2}\\

The fifth conjunct in the \textit{LL1Init} predicate relates to the \textit{LL1ObservedOutputs} variable. The proof is straightforward, because the equality is direct.

\\[\text{(2)11. LL1ObservedOutputs} = \{\}\\]

\textbf{(3)1. LL2ObservedOutputs} = \{\}\\

\textbf{By (2)2}\\

\textbf{(3)2. LL1ObservedOutputs} = \text{LL2ObservedOutputs}\\

\textbf{By (2)1 Def LL2Refinement}\\

\textbf{(3)3. QED}\\

\textbf{By (3)1, (3)2}\\

The sixth conjunct in the \textit{LL1Init} predicate relates to the \textit{LL1ObservedAuthenticators} variable. There are four steps: (1) The refinement asserts that the corresponding fields from the two specs match according to the \textit{HistorySummariesMatch} predicate. (2) We prove that the initial authenticator values match across the two specs. (3) We prove that the Memoir-Opt set of observed authenticators equals its initial value. (4) We use the \textit{AuthenticatorSetsMatchUniqueLemma} to prove that the variable in the Memoir-Basic spec must equal the initial value.

\\[\text{(2)12. LL1ObservedAuthenticators} = \{\text{ll1InitialAuthenticator}\}\\]

The corresponding variables from the two specs match, thanks to the refinement.

\textbf{(3)1. AuthenticatorSetsMatch(}\\
\textit{LL1ObservedAuthenticators,}\\
\textit{LL2ObservedAuthenticators,}\\
\textit{LL2NVRAM.symmetricKey,}\\
\textit{LL2NVRAM.hashBarrier)}\\

\textbf{By (2)1 Def LL2Refinement}\\

The corresponding sets of initial authenticator values match across the two specs. Since the sets are singletons, this follows trivially from proving that one element in each set matches the other. And we proved that these elements match above.

\textbf{(3)2. AuthenticatorSetsMatch(}\\
\textit{\{ll1InitialAuthenticator\},}\\
\textit{\{ll2InitialAuthenticator\},}\\
\textit{LL2NVRAM.symmetricKey,}\\
\textit{LL2NVRAM.hashBarrier)}\\

\textbf{(4)1. AuthenticatorsMatch(}\\
\textit{ll1InitialAuthenticator,}\\
\textit{ll2InitialAuthenticator,}\\
\textit{LL2NVRAM.symmetricKey,}\\
\textit{LL2NVRAM.hashBarrier)}\\

\textbf{By (2)6}\\

\textbf{(4)2. QED}
The Memoir-Opt spec’s set of observed authenticators equals its initial authenticator value, as asserted by the \( LL2Init \) predicate.

\( LL2ObservedAuthenticators = \{ ll2InitialAuthenticator \} \)

We use the \textit{AuthenticatorSetsMatchUniqueLemma} to prove that the variable in the Memoir-Basic spec equals its initial value. This requires proving some types.

\( LL2ObservedAuthenticators \in \text{subset MACType} \)

The conjunction of the above six conjuncts.

For the induction step, we will need the refinement and the type invariant to be true in both the unprimed and primed states.

We assume the antecedents.

We then prove that each step in the Memoir-Opt spec refines to a step in the Memoir-Basic spec. First, a Memoir-Opt stuttering step refines to a Memoir-Basic stuttering step. We prove the unchanged status for each Memoir-Basic variable in turn, using the lemmas we have proven for this purpose.
We prove each conjunct in the \textit{LL1 Make Input Available} action separately.

\begin{enumerate}
\item \textbf{input} \notin \textit{LL1 Available Inputs}
\begin{enumerate}
\item \textbf{input} \notin \textit{LL2 Available Inputs}
\begin{enumerate}
\item \textit{LL1 Available Inputs} = \textit{LL2 Available Inputs}
\begin{enumerate}
\item \textit{LL1 Available Inputs}' = \textit{LL2 Available Inputs}'
\end{enumerate}
\end{enumerate}
\item \textbf{QED}
\begin{enumerate}
\item \textbf{By (3) 2, (4) 2, (4) 3}
\item \textit{LL1 Available Inputs}' = \textit{LL1 Available Inputs} \cup \{ \textbf{input} \}
\begin{enumerate}
\item \textit{LL2 Available Inputs}' = \textit{LL2 Available Inputs} \cup \{ \textbf{input} \}
\begin{enumerate}
\item \textbf{By (3) 2}
\item \textit{LL1 Available Inputs} = \textit{LL2 Available Inputs}
\begin{enumerate}
\item \textbf{By (2) 1 DEF LL2 Refinement}
\item \textit{LL1 Available Inputs}' = \textit{LL2 Available Inputs}'
\begin{enumerate}
\item \textbf{By (2) 1 DEF LL2 Refinement}
\item \textbf{QED}
\begin{enumerate}
\item \textbf{By (4) 1, (4) 2, (4) 3}
\end{enumerate}
\item \textbf{By (2) 1, (3) 2, Unchanged Disk Lemma}
\item \textbf{By (2) 1, (3) 2, Unchanged RAM Lemma}
\item \textbf{By (2) 1, (3) 2, Unchanged NVRAM Lemma}
\item \textbf{By (2) 1, (3) 2, Unchanged Observed Outputs Lemma}
\item \textbf{By (2) 1, (3) 2, Unchanged Observed Authenticators Lemma}
\item \textbf{By (3) 3, (3) 4, (3) 5, (3) 7, (3) 8, (3) 9 DEF LL1 Make Input Available}
\end{enumerate}
\end{enumerate}
\end{enumerate}
\end{enumerate}
\end{enumerate}
\end{enumerate}
\end{enumerate}
We prove that the definitions from \texttt{LL2PerformOperation} satisfy their types, using the \texttt{LL2PerformOperationDefsTypeSafeLemma}.

\begin{itemize}
  \item \texttt{ll2HistorySummaryHash} \triangleq \texttt{Hash(\text\texttt{LL2RAM.historySummary.anchor}, \text\texttt{LL2RAM.historySummary.extension})}
  \item \texttt{ll2StateHash} \triangleq \texttt{Hash(\text\texttt{LL2RAM.publicState}, \text\texttt{LL2RAM.privateStateEnc})}
  \item \texttt{ll2HistoryStateBinding} \triangleq \texttt{Hash(\text\texttt{ll2HistorySummaryHash, ll2StateHash})}
  \item \texttt{ll2PrivateKey} \triangleq \texttt{\text\texttt{SymmetricDecrypt(\text\texttt{LL2NVRAM.symmetricKey, ll2PrivateKeyEnc})}}
  \item \texttt{ll2HistorySummary} \triangleq \texttt{\text\texttt{Service(\text\texttt{LL2RAM.publicState, ll2PrivateKey, input}})
  \item \texttt{ll2NewPrivateKeyEnc} \triangleq \texttt{\text\texttt{SymmetricEncrypt(\text\texttt{LL2NVRAM.symmetricKey, ll2Result.newPrivateState})}}
  \item \texttt{ll2CurrentHistorySummary} \triangleq \texttt{\text\texttt{[\text\texttt{anchor} \text\texttt{\rightarrow LL2NVRAM.historySummaryAnchor, extension} \text\texttt{\rightarrow LL2SPCR]}}
  \item \texttt{ll2NewHistorySummary} \triangleq \texttt{\text\texttt{Successor(ll2CurrentHistorySummary, input, LL2NVRAM.hashBarrier}})
  \item \texttt{ll2NewHistorySummaryHash} \triangleq \texttt{\text\texttt{Hash(ll2NewHistorySummary.anchor, ll2NewHistorySummary.extension)}}
  \item \texttt{ll2NewStateHash} \triangleq \texttt{\text\texttt{Hash(ll2Result.newPublicState, ll2NewPrivateStateEnc)}}
  \item \texttt{ll2NewHistoryStateBinding} \triangleq \texttt{\text\texttt{Hash(ll2NewHistorySummaryHash, ll2NewStateHash)}}
  \item \texttt{ll2NewAuthenticator} \triangleq \texttt{\text\texttt{GenerateMAC(\text\texttt{LL2NVRAM.symmetricKey, ll2NewHistoryStateBinding})}}
\end{itemize}
We hide the definitions.

We restate the definitions from LL1PerformOperation.

We prove that the definitions from LL1PerformOperation satisfy their types, using the LL1PerformOperationDefsTypeSafeLemma and the TypeSafetyRefinementLemma.

We hide the definitions from LL1PerformOperation.

We prove the correspondences between the definitions in LL1PerformOperation and LL2PerformOperation. The state hashes are directly equal.
(3.7) \( \mathit{ll1PrivateState} = \mathit{ll2PrivateState} \)
   (4.1) \( \mathit{LL1NVRAM}.\mathit{symmetricKey} = \mathit{LL2NVRAM}.\mathit{symmetricKey} \)
   BY (2) 1 DEF \( \mathit{LL2Refinement} \)
   (4.2) \( \mathit{LL1RAM}.\mathit{privateStateEnc} = \mathit{LL2RAM}.\mathit{privateStateEnc} \)
   BY (2) 1 DEF \( \mathit{LL2Refinement} \)
   (4.3) QED
   BY (4.1), (4.2) DEF \( \mathit{ll1StateHash}, \mathit{ll2StateHash} \)

The private states are directly equal across the two specs.

(3.8) \( \mathit{ll1sResult} = \mathit{ll2SResult} \)
   (4.1) \( \mathit{LL1RAM}.\mathit{publicState} = \mathit{LL2RAM}.\mathit{publicState} \)
   BY (2) 1 DEF \( \mathit{LL2Refinement} \)
   (4.2) \( \mathit{ll1PrivateState} = \mathit{ll2PrivateState} \)
   BY (3) 7
   (4.3) QED
   BY (4.1), (4.2) DEF \( \mathit{ll1sResult}, \mathit{ll2SResult} \)

The service results are directly equal across the two specs.

(3.9) \( \mathit{ll1NewPrivateStateEnc} = \mathit{ll2NewPrivateStateEnc} \)
   (4.1) \( \mathit{LL1NVRAM}.\mathit{symmetricKey} = \mathit{LL2NVRAM}.\mathit{symmetricKey} \)
   BY (2) 1 DEF \( \mathit{LL2Refinement} \)
   (4.2) \( \mathit{ll1sResult}.\mathit{newPrivateState} = \mathit{ll2SResult}.\mathit{newPrivateState} \)
   BY (3) 8
   (4.3) QED
   BY (4.1), (4.2) DEF \( \mathit{ll1NewPrivateStateEnc}, \mathit{ll2NewPrivateStateEnc} \)

The new encrypted private states are directly equal across the two specs.

(3.10) \( \mathit{HistorySummariesMatch}( \mathit{ll1NewHistorySummary}, \mathit{ll2NewHistorySummary}, \mathit{LL2NVRAM}.\mathit{hashBarrier}') \)

First, we prove that the \( \mathit{HistorySummariesMatch} \) predicate equals the \( \mathit{HistorySummariesMatchRecursion} \) predicate in this case. We assert each condition required by the definition of the predicate.

(4.1) \( \mathit{HistorySummariesMatch}( \mathit{ll1NewHistorySummary}, \mathit{ll2NewHistorySummary}, \mathit{LL2NVRAM}.\mathit{hashBarrier}') = \mathit{HistorySummariesMatchRecursion}( \mathit{ll1NewHistorySummary}, \mathit{ll2NewHistorySummary}, \mathit{LL2NVRAM}.\mathit{hashBarrier}') \)

We begin by proving the types for the \( \mathit{HistorySummariesMatchRecursion} \) predicate.

(5.1) \( \mathit{ll1NewHistorySummary} \in \mathit{HashType} \)
   BY (3) 5
(5.2) \( \mathit{ll2NewHistorySummary} \in \mathit{HistorySummaryType} \)
   BY (3) 3
(5.3) \( \mathit{LL2NVRAM}.\mathit{hashBarrier}' \in \mathit{HashType} \)
   BY (2) 1, \( \mathit{LL2subtypeImplicationLemma} \) DEF \( \mathit{LL2subtypeImplication} \)
(5) \( \mathit{ll2InitialHistorySummary} \triangleq [\mathit{anchor} \mapsto \mathit{BaseHashValue}, \mathit{extension} \mapsto \mathit{BaseHashValue}] \)

We then prove that this is not the base case for the \( \mathit{HistorySummariesMatch} \) predicate.

(5.4) \( \mathit{ll2NewHistorySummary} \neq \mathit{ll2InitialHistorySummary} \)

This proof has a lot of sub-steps, but it is pretty simple. We just use the \( \mathit{BaseHashValueUnique} \) property to show that the extension field in the \( \mathit{ll2NewHistorySummary} \) record cannot match the base hash value, which is the value of the extension field in the initial history summary.
6.1. \( \text{l2NewHistorySummary} = \) 
\( \text{Successor(\text{l2CurrentHistorySummary}, \text{input}, \text{LL2NVRAM}.\text{hashBarrier})} \)

BY DEF \( \text{l2NewHistorySummary} \)

6.2. \( \text{l2NewHistorySummary}.\text{extension} = \) 
\( \text{Hash(\text{l2CurrentHistorySummary}.\text{extension}, \text{Hash(\text{LL2NVRAM}.\text{hashBarrier}, \text{input})})} \)

BY (6.1) DEF Successor

6.3. \( \text{l2NewHistorySummary}.\text{extension} \neq \text{BaseHashValue} \)

7.1. \( \text{l2CurrentHistorySummary}.\text{extension} \in \text{HashDomain} \)

8.1. \( \text{l2CurrentHistorySummary}.\text{extension} \in \text{HashType} \)

9.1. \( \text{l2CurrentHistorySummary}.\text{extension} = \text{LL2SPCR} \)

BY DEF \( \text{l2CurrentHistorySummary} \)

9.2. \( \text{LL2SPCR} \in \text{HashType} \)

BY (2.1) DEF LL2TypeInvariant

9.3. QED

BY (9.1), (9.2)

8.2. QED

BY (8.1) DEF HashDomain

7.2. \( \text{Hash(\text{LL2NVRAM}.\text{hashBarrier}, \text{input})} \in \text{HashDomain} \)

8.1. \( \text{Hash(\text{LL2NVRAM}.\text{hashBarrier}, \text{input})} \in \text{HashType} \)

9.1. \( \text{LL2NVRAM}.\text{hashBarrier} \in \text{HashDomain} \)

10.1. \( \text{LL2NVRAM}.\text{hashBarrier} \in \text{HashType} \)

BY (2.1), LL2SubtypeImplicationLemma DEF LL2SubtypeImplicationLemma

10.2. QED

BY (10.1) DEF HashDomain

9.2. \( \text{input} \in \text{HashDomain} \)

10.1. \( \text{input} \in \text{InputType} \)

11.1. \( \text{input} \in \text{LL2AvailableInputs} \)

BY (3.2)

11.2. \( \text{LL2AvailableInputs} \subseteq \text{InputType} \)

BY (2.1) DEF LL2TypeInvariant

11.3. QED

BY (11.1), (11.2)

10.2. QED

BY (10.1) DEF HashDomain

9.3. QED

BY (9.1), (9.2), HashTypeSafe

8.2. QED

BY (8.1) DEF HashDomain

7.3. QED

BY (6.2), (7.1), (7.2), BaseHashValueUnique

6.4. QED

BY (6.3)

Since this is not the base case, the \( \text{HistorySummariesMatch} \) predicate equals the \( \text{HistorySummariesMatchRecursion} \) predicate.

5.5. QED

BY (5.1), (5.2), (5.3), (5.4), HistorySummariesMatchDefinition

Then, we prove that the \( \text{HistorySummariesMatchRecursion} \) predicate is satisfied. We assert each condition required by the definition of the predicate.

4.2. \( \text{HistorySummariesMatchRecursion(} \)
\( \text{l1NewHistorySummary, l2NewHistorySummary, LL2NVRAM}.\text{hashBarrier}) \)

We begin by proving the types for the existentially quantified variables in the \( \text{HistorySummariesMatchRecursion} \) predicate.

298
(5.1) \( \text{input} \in \text{InputType} \)

(6.1) \( \text{input} \in \text{LL2AvailableInputs} \)

\begin{align*}
\text{by} & \ (3)2 \\
\text{(6.2)} & \ \text{LL2AvailableInputs} \subseteq \text{InputType} \\
\text{by} & \ (2)1 \ \text{def \ LL2TypeInvariant} \\
\text{(6.3)} & \ \text{QED}
\end{align*}

\begin{align*}
\text{by} & \ (6.1), (6.2) \\
(5.2) & \ \text{LL1NVRAM.historySummary} \in \text{HashType} \\
\text{by} & \ (2)1 \ \text{def \ LL2Refinement, LL1TrustedStorageType} \\
(5.3) & \ \text{LL2NVRAMLogicalHistorySummary} \in \text{HistorySummaryType} \\
\text{by} & \ (2)1, \ \text{LL2NVRAMLogicalHistorySummaryTypeSafe}
\end{align*}

We then prove the three conjuncts in the \( \text{HistorySummariesMatchRecursion} \) predicate. The first conjunct follows directly from the refinement.

\begin{align*}
\text{(5.4)} & \ \text{HistorySummariesMatch(} \\
& \quad \text{LL1NVRAM.historySummary,} \\
& \quad \text{LL2NVRAMLogicalHistorySummary,} \\
& \quad \text{LL2NVRAM.hashBarrier'}) \\
\text{(6.1)} & \ \text{unchanged LL2NVRAM.hashBarrier} \\
\text{by} & \ (3)2 \\
\text{(6.2)} & \ \text{QED}
\end{align*}

\text{by} \ (2)1, (6.1) \ \text{def \ LL2Refinement}

The second conjunct in the \( \text{HistorySummariesMatchRecursion} \) predicate is true by definition.

\begin{align*}
\text{(5.5)} & \ \text{ll1NewHistorySummary} = \text{Hash(LL1NVRAM.historySummary, input)} \\
\text{by \ def \ ll1NewHistorySummary}
\end{align*}

The third conjunct in the \( \text{HistorySummariesMatchRecursion} \) predicate is true because the \( \text{ll2NewHistorySummary} \) definition in the \( \text{LL2PerformOperation} \) action yields a value that is the successor of the logical history summary in the NVRAM.

\begin{align*}
\text{(5.6)} & \ \text{LL2HistorySummaryIsSuccessor(} \\
& \quad \text{ll2NewHistorySummary,} \\
& \quad \text{LL2NVRAMLogicalHistorySummary,} \\
& \quad \text{input,} \\
& \quad \text{LL2NVRAM.hashBarrier'}) \\
\text{(6.1)} & \ \text{LL2NVRAMLogicalHistorySummary} = \text{ll2CurrentHistorySummary} \\
\text{(7.1)} & \ \text{case \ LL2NVRAM.extensionInProgress = \text{TRUE}} \\
\text{(8.2)} & \ \text{LL2NVRAMLogicalHistorySummary} = [ \\
& \quad \text{anchor} \mapsto \text{LL2NVRAM.historySummaryAnchor}, \\
& \quad \text{extension} \mapsto \text{LL2SPCR}] \\
\text{(9.1)} & \ \text{LL2SPCR} \neq \text{BaseHashValue} \\
\text{by} & \ (3)2, (7)1 \\
\text{(9.2)} & \ \text{QED} \\
\text{by} & \ (7)1, (9.1) \ \text{def \ LL2NVRAMLogicalHistorySummary} \\
\text{(8.3)} & \ \text{ll2CurrentHistorySummary} = [ \\
& \quad \text{anchor} \mapsto \text{LL2NVRAM.historySummaryAnchor}, \\
& \quad \text{extension} \mapsto \text{LL2SPCR}] \\
\text{by \ def \ ll2CurrentHistorySummary} \\
\text{(8.4)} & \ \text{QED} \\
\text{by} & \ (8)2, (8)3 \\
\text{(7.2)} & \ \text{case \ LL2NVRAM.extensionInProgress = \text{FALSE}} \\
\text{(8.1)} & \ \text{LL2NVRAMLogicalHistorySummary} = [ \\
& \quad \text{anchor} \mapsto \text{LL2NVRAM.historySummaryAnchor}, \\
& \quad \text{extension} \mapsto \text{BaseHashValue}] \\
\text{by} & \ (7)2 \ \text{def \ LL2NVRAMLogicalHistorySummary}\)
The new authenticators match across the two specs, as specified by the AuthenticatorsMatch predicate. First, we prove some types needed by the definition of the AuthenticatorsMatch predicate.

\[ ]
\begin{align*}
\text{ll2CurrentHistorySummary} &= [ \\
&\quad \text{anchor} \mapsto \text{LL2NVRAM.historySummary.Anchor}, \\
&\quad \text{extension} \mapsto \text{LL2SPCR}]
\end{align*}
\]

\text{ll2CurrentHistorySummary}

\[ ]
\begin{align*}
\text{ll2SPCR} &= \text{BaseHashValue} \\
\text{by} (3)2, (7)2
\end{align*}
\]

\text{ll2SPCR} = \text{BaseHashValue}

\[ ]
\begin{align*}
\text{ll2NewHistorySummary} &= \\
&\quad \text{Successor}(\text{LL2NVRAMLogicalHistorySummary}, \text{input}, \text{LL2NVRAM.hashBarrier})
\end{align*}
\]

\text{ll2NewHistorySummary} = \text{Successor}(\text{LL2NVRAMLogicalHistorySummary}, \text{input}, \text{LL2NVRAM.hashBarrier})

\[ ]
\begin{align*}
&\text{ll2NewHistorySummary} \\
&\text{by (6)1 DEF ll2NewHistorySummary} \\
&\text{ll2NewHistorySummary} \\
&\text{by (6)2, (6)3 DEF LL2HistorySummaryIsSuccessor}
\end{align*}
\]

All conjuncts in the HistorySummariesMatchRecursion predicate are satisfied.

\[ ]
\begin{align*}
&\text{ll2NewStateHash} = \text{ll2NewStateHash} \\
&\text{by (3)11} \\
&\text{ll1NewStateHash} = \text{ll2NewStateHash} \\
&\text{by (4)1} \\
&\text{ll1NewStateHash} = \text{ll2NewStateHash}
\end{align*}
\]

The new state hashes are directly equal across the two specs.

\[ ]
\begin{align*}
&\text{ll2NewStateHash} \in \text{HashType} \\
&\text{by (3)3} \\
&\text{ll1NewHistorySummary} \in \text{HashType} \\
&\text{by (3)5} \\
&\text{ll2NewHistorySummary} \in \text{HistorySummaryType} \\
&\text{by (3)3}
\end{align*}
\]

We then prove that, in the Memoir-Opt spec, the new authenticator is a valid MAC for the new history state binding. We will use the MACComplete property.

\[ ]
\begin{align*}
&\text{ValidateMAC(} \text{LL2NVRAM.symmetricKey'}, \text{ll2NewHistoryStateBinding}, \text{ll2NewAuthenticator}) \\
&\text{In the Memoir-Opt spec, the new authenticator is generated as a MAC of the new history state binding.}
\end{align*}
\]

300
(5.1) \texttt{ll2NewAuthenticator} =
\begin{align*}
\text{GenerateMAC}(\text{ll2NVRAM}.\text{symmetricKey}, \text{ll2NewHistoryStateBinding})
\end{align*}

(6.1) UNCHANGED \texttt{ll2NVRAM}.\text{symmetricKey}
\begin{align*}
\text{by (3)2}
\end{align*}

(6.2) QED
\begin{align*}
\text{by (6)1 def ll2NewAuthenticator}
\end{align*}

We can thus use the MACComplete property to show that the generated MAC validates appropriately. To do this, we first need to prove some types.

(5.2) \texttt{ll2NVRAM}.\text{symmetricKey} \in \text{SymmetricKeyType}
\begin{align*}
\text{by (2)1, ll2SubtypeImplicationLemma def ll2SubtypeImplication}
\end{align*}

(5.3) \texttt{ll2NewHistoryStateBinding} \in \text{HashType}
\begin{align*}
\text{by (3)3}
\end{align*}

Then, we appeal to the MACComplete property in a straightforward way.

(5.4) QED
\begin{align*}
\text{by (5)1, (5)2, (5)3, MACComplete}
\end{align*}

We then prove that, in the Memoir-Basic spec, the new authenticator is generated as a MAC of the new history state binding.

(4.5) \texttt{ll1NewAuthenticator} = \text{GenerateMAC}(\text{ll1NVRAM}.\text{symmetricKey}, \text{ll1NewHistoryStateBinding})
\begin{align*}
(5.1) \text{ll1NVRAM}.\text{symmetricKey} = \text{ll2NVRAM}.\text{symmetricKey}
\end{align*}

(6.1) \text{ll1NVRAM}.\text{symmetricKey} = \text{ll2NVRAM}.\text{symmetricKey}
\begin{align*}
\text{by (2)1 def ll2Refinement}
\end{align*}

(6.2) UNCHANGED \text{ll2NVRAM}.\text{symmetricKey}
\begin{align*}
\text{by (3)2}
\end{align*}

(6.3) QED
\begin{align*}
\text{by (6)1, (6)2}
\end{align*}

(5.2) QED
\begin{align*}
\text{by (5)1 def ll1NewAuthenticator}
\end{align*}

The new history summaries match across the two specs, as we proved above.

(4.6) HistorySummariesMatch(
\begin{align*}
\text{ll1NewHistorySummary, ll2NewHistorySummary, ll2NVRAM}.\text{hashBarrier}
\end{align*}
\begin{align*}
\text{by (3)10}
\end{align*}

We then invoke the definition of the AuthenticatorsMatch predicate.

(4.7) QED
\begin{align*}
\text{by (3)11, (4)1, (4)2, (4)3, (4)4, (4)5, (4)6}
\end{align*}

\text{def AuthenticatorsMatch, ll1NewHistoryStateBinding, ll2NewHistoryStateBindingHash, ll2NewHistoryStateBinding}

The remainder of the proof for LL1PerformOperation is a series of assertions, one for each conjunct in the definition of the LL1PerformOperation action.

The first conjunct in LL1PerformOperation. This is basically just an application of AuthenticatorValidatedLemma

(3.13) ValidateMAC(\text{ll1NVRAM}.\text{symmetricKey}, \text{ll1HistoryStateBinding}, \text{ll1RAM}.\text{authenticator})
\begin{align*}
\text{We need the fact that the symmetric keys in the NVRAM are equal across the two specs.}
\end{align*}

(4.1) \text{ll1NVRAM}.\text{symmetricKey} = \text{ll2NVRAM}.\text{symmetricKey}
\begin{align*}
\text{by (2)1 def ll2Refinement}
\end{align*}

We prove the types that are needed for AuthenticatorValidatedLemma.

(4.2) \text{ll2StateHash} \in \text{HashType}
\begin{align*}
\text{by (3)3}
\end{align*}

(4.3) \text{ll1RAM}.\text{historySummary} \in \text{HashType}
\begin{align*}
\text{by (2)1 def ll2Refinement, ll1UntrustedStorageType}
\end{align*}

(4.4) \text{ll2RAM}.\text{historySummary} \in \text{HistorySummaryType}
\begin{align*}
\text{by (2)1, ll2SubtypeImplicationLemma def ll2SubtypeImplication}
\end{align*}
(4.5). \( \text{LL1RAM.authenticator} \in \text{MACType} \)

BY (2.1) DEF LL2Refinement, LL1UntrustedStorageType

(4.6). \( \text{LL2RAM.authenticator} \in \text{MACType} \)

BY (2.1), LL2SubtypeImplicationLemma DEF LL2SubtypeImplication

(4.7). \( \text{LL2NVRAM.symmetricKey} \in \text{SymmetricKeyType} \)

BY (2.1), LL2SubtypeImplicationLemma DEF LL2SubtypeImplication

(4.8). \( \text{LL2NVRAM.hashBarrier} \in \text{HashType} \)

BY (2.1), LL2SubtypeImplicationLemma DEF LL2SubtypeImplication

There are three preconditions for AuthenticatorValidatedLemma. The first precondition follows from the refinement.

(4.9). \( \text{HistorySummariesMatch(} \)

\( \text{LL1RAM.historySummary, LL2RAM.historySummary, LL2NVRAM.hashBarrier} \)

BY (2.1) DEF LL2Refinement

The second precondition also follows from the refinement.

(4.10). \( \text{PICK symmetricKey} \in \text{SymmetricKeyType : AuthenticatorsMatch(} \)

\( \text{LL1RAM.authenticator, LL2RAM.authenticator, symmetricKey, LL2NVRAM.hashBarrier} \)

BY (2.1) DEF LL2Refinement

The third precondition follows from the definition of the \( \text{LL2PerformOperation} \) action.

(4.11). \( \text{ValidateMAC(} \text{LL2NVRAM.symmetricKey, ll2HistoryStateBinding, LL2RAM.authenticator} \)

BY (3.2) DEF ll2HistoryStateBinding, ll2StateHash, ll2HistorySummaryHash

(4.12). QED

Ideally, this QED step should just read:

BY (3.6), (4.1), (4.2), (4.3), (4.4), (4.5), (4.6), (4.7), (4.8), (4.9), (4.10), (4.11),

AuthenticatorValidatedLemma!

DEF ll1HistoryStateBinding, ll2HistoryStateBinding, ll2HistorySummaryHash

However, the prover seems to get a little confused in this instance. We make life easier for the prover by explicitly staging the instantiation of the quantified variables within the definition of AuthenticatorValidatedLemma.

(5.1). \( \forall \text{samLL2Authenticator} \in \text{MACType}, \)

\( \text{samSymmetricKey2} \in \text{SymmetricKeyType}, \)

\( \text{samHashBarrier} \in \text{HashType} : \)

AuthenticatorValidatedLemma!(ll2StateHash, LL1RAM.historySummary, 

\( \text{LL2RAM.historySummary, LL1RAM.authenticator, samLL2Authenticator, symmetricKey, samSymmetricKey2, samHashBarrier})!1 \)

BY (4.2), (4.3), (4.4), (4.5), AuthenticatorValidatedLemma

(5.2). \( \text{AuthenticatorValidatedLemma!(ll2StateHash, LL1RAM.historySummary, LL2RAM.historySummary, LL1RAM.authenticator, LL2RAM.authenticator,} \)

\( \text{symmetricKey, LL2NVRAM.symmetricKey, LL2NVRAM.hashBarrier})!1 \)

BY (4.2), (4.3), (4.4), (4.5), (4.6), (4.7), (4.8), (4.10), (5.1)

(5.3). QED

BY (3.6), (4.1), (4.2), (4.3), (4.4), (4.5), (4.6), (4.7), (4.8), (4.9), (4.10), (4.11), (5.2)

DEF ll1HistoryStateBinding, ll2HistoryStateBinding, ll2HistorySummaryHash

The second conjunct in \( \text{LL1PerformOperation} \): In the Memoir-Basic spec, the history summary in the NVRAM

(3.14). \( \text{LL1NVRAM.historySummary} = \text{LL1RAM.historySummary} \)

The history summaries in the NVRAM match across the two specs.

(4.1). HistorySummariesMatch(
NVRAM
If an extension is not in progress, then the logical history summary in the
NVRAM
is in progress, then the logical history summary in the
NVRAM
In the Memoir-Opt spec, we separately handle the two cases of whether an extension is in progress. If an extension is in progress, the history summaries in the RAM match across the two specs.

We use the HistorySummariesMatchUniqueLemma to prove that the fields are equal. This requires proving some types.

If an extension is not in progress, then the logical history summary in the NVRAM is a checkpoint of the history summary in the RAM, so we can use the HistorySummariesMatchAcrossCheckpointLemma.

(4)2. HistorySummariesMatch(
    LL1RAM.historySummary, LL2RAM.historySummary, LL2NVRAM.hashBarrier)
    BY (2)1 DEF LL2Refinement

The history summaries in the RAM match across the two specs.

(4)3. CASE LL2NVRAM.extensionInProgress
    (5)1. LL2NVRAMLogicalHistorySummary = LL2RAM.historySummary
        (6)1. LL2NVRAMLogicalHistorySummary = ll2CurrentHistorySummary
            (7)1. LL2NVRAMLogicalHistorySummary = [
                anchor ↦ LL2NVRAM.historySummaryAnchor,
                extension ↦ LL2SPCR]
            ⟨8⟩1. LL2SPCR ≠ BaseHashValue
                BY (3)2, (4)3
                ⟨8⟩2. QED
                BY (4)3, ⟨8⟩1 DEF LL2NVRAMLogicalHistorySummary
            ⟨7⟩2. ll2CurrentHistorySummary = [
                anchor ↦ LL2NVRAM.historySummaryAnchor,
                extension ↦ LL2SPCR]
                BY DEF ll2CurrentHistorySummary
                ⟨7⟩3. QED
                BY ⟨7⟩1, ⟨7⟩2
            ⟨6⟩2. ll2CurrentHistorySummary = LL2RAM.historySummary
                BY ⟨3⟩2, ⟨4⟩3 DEF ll2CurrentHistorySummary
                ⟨6⟩3. QED
                BY ⟨6⟩1, ⟨6⟩2
        ⟨6⟩1. LL1NVRAM.historySummary ∈ HashType
            BY ⟨2⟩1 DEF LL2Refinement, LL1TrustedStorageType
        ⟨6⟩2. LL1RAM.historySummary ∈ HashType
            BY ⟨2⟩1 DEF LL2Refinement, LL1UntrustedStorageType
        ⟨6⟩3. LL2RAM.historySummary ∈ HistorySummaryType
            BY ⟨2⟩1, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication
        ⟨6⟩4. LL2NVRAM.hashBarrier ∈ HashType
            BY ⟨2⟩1, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication
        ⟨6⟩5. QED
            BY ⟨4⟩1, ⟨4⟩2, ⟨5⟩1, ⟨6⟩1, ⟨6⟩2, ⟨6⟩3, ⟨6⟩4, HistorySummariesMatchUniqueLemma

We use the HistorySummariesMatchUniqueLemma to prove that the fields are equal. This requires proving some types.

(4)4. CASE ¬LL2NVRAM.extensionInProgress
    (5)1. LL2NVRAMLogicalHistorySummary = Checkpoint(LL2RAM.historySummary)
        (6)1. LL2NVRAMLogicalHistorySummary = ll2CurrentHistorySummary
            (7)1. LL2NVRAMLogicalHistorySummary = [
                anchor ↦ LL2NVRAM.historySummaryAnchor,
                extension ↦ BaseHashValue]
We use the HistorySummariesMatchAcrossCheckpointLemma to prove that the fields are equal. This requires proving some types.

(5)2. QED
(6)1. LL1NVRAM.historySummary ∈ HashType
   BY (2)1 DEF LL2Refinement, LL1TrustedStorageType
(6)2. LL1RAM.historySummary ∈ HashType
   BY (2)1 DEF LL2Refinement, LL1UntrustedStorageType
(6)3. LL2NVRAMLLogicalHistorySummary ∈ HistorySummaryType
   BY (2)1, LL2NVRAMLLogicalHistorySummaryTypeSafe
(6)4. LL2RAM.historySummary ∈ HistorySummaryType
   BY (2)1, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication
(6)5. LL2NVRAM.hashBarrier ∈ HashType
   BY (2)1, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication
(6)6. QED
   BY (4)1, (4)2, (5)1, (6)1, (6)2, (6)3, (6)4, (6)5, HistorySummariesMatchAcrossCheckpointLemma
(4)5. QED
   BY (4)3, (4)4
The third conjunct in LL1PerformOperation. This conjunct asserts a record equality, so we prove each field in the record separately.

(3)15. LL1RAM' = [
   publicState ↦ ll1sResult.newPublicState,
   privateStateEnc ↦ ll1NewPrivateKeyEnc,
   historySummary ↦ ll1NewHistorySummary,
   authenticator ↦ ll1NewAuthenticator]

The public state field in the primed Memoir-Basic RAM equals the public state in the result of the service, because the primed RAM variables match across the two specs.

(4)1. LL1RAM.publicState' = ll1sResult.newPublicState
(5)1. LL1RAM.publicState' = LL2RAM.publicState'
   BY (2)1 DEF LL2Refinement
(5)2. LL2RAM.publicState' = ll2sResult.newPublicState
   BY (3)2 DEF ll2sResult, ll2sPrivateState
(5)3. ll1sResult.newPublicState = ll2sResult.newPublicState
   BY (3)8
(5)4. QED
   BY (5)1, (5)2, (5)3
The encrypted private state field in the primed Memoir-Basic RAM equals the encrypted private state in the result of the service, because the primed RAM variables match across the two specs.

(4)2. LL1RAM.privateStateEnc' = ll1NewPrivateKeyEnc
(5)1. LL1RAM.privateStateEnc' = LL2RAM.privateStateEnc'
The authenticator field in the primed Memoir-Basic RAM equals the new authenticator defined in the HistorySummariesMatchUniqueLemma refinement. We use the HistorySummariesMatchUniqueLemma to prove the equality.

By (2)1 DEF LL2Refinement
(5)2. \text{LL2RAM}.privateStateEnc' = ll2NewPrivateStateEnc
By (3)2 DEF ll2NewPrivateStateEnc, ll2SResult, ll2PrivateState
(5)3. ll1NewPrivateStateEnc = ll2NewPrivateStateEnc
By (3)9
(5)4. QED
By (5)1, (5)2, (5)3

The history summary field in the primed Memoir-Basic RAM equals the new history summary defined in the PerformOperation action, because the history summaries in the RAM variables match across the specs by refinement. We use the HistorySummariesMatchUniqueLemma to prove the equality.

(4)3. ll1RAM.historySummary' = ll1NewHistorySummary
(5)1. HistorySummariesMatch(
    ll1RAM.historySummary', ll2RAM.historySummary', ll2NVRAM.hashBarrier')
By (2)1 DEF LL2Refinement
(5)2. ll2RAM.historySummary' = ll2NewHistorySummary
By (3)2 DEF ll2NewHistorySummary, ll2CurrentHistorySummary
(5)3. HistorySummariesMatch(
    ll1NewHistorySummary, ll2NewHistorySummary, ll2NVRAM.hashBarrier')
By (3)10
We use the HistorySummariesMatchUniqueLemma to prove that the fields are equal. This requires proving some types.

(5)4. QED
(6)1. ll1RAM.historySummary' \in HashType
By (2)1 DEF LL2Refinement, LL1UntrustedStorageType
(6)2. ll1NewHistorySummary \in HashType
By (3)5
(6)3. ll2RAM.historySummary' \in HistorySummaryType
By (2)1, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication
(6)4. ll2NVRAM.hashBarrier' \in HashType
By (2)1, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication
(6)5. QED
By (5)1, (5)2, (5)3, (6)1, (6)2, (6)3, (6)4, HistorySummariesMatchUniqueLemma

The authenticator field in the primed Memoir-Basic RAM equals the new authenticator defined in the PerformOperation action, because the authenticators in the RAM variables match across the specs by refinement. We use the AuthenticatorsMatchUniqueLemma to prove the equality.

(4)4. ll1RAM.authenticator' = ll1NewAuthenticator
(5)1. PICK symmetricKey \in SymmetricKeyType:
    AuthenticatorsMatch(
        ll1RAM.authenticator',
        ll2RAM.authenticator',
        symmetricKey,
        ll2NVRAM.hashBarrier'
    )
By (2)1 DEF LL2Refinement
(5)2. ll2RAM.authenticator' = ll2NewAuthenticator
By (3)2
DEF ll2NewAuthenticator, ll2NewHistoryStateBinding, ll2NewHistorySummaryHash,
ll2NewStateHash, ll2NewPrivateStateEnc, ll2SResult, ll2PrivateState,
ll2NewHistorySummary, ll2CurrentHistorySummary
(5)3. AuthenticatorsMatch(
    ll1NewAuthenticator,
    ll2NewAuthenticator,
    ll2NVRAM.symmetricKey',
)
We use the $AuthenticatorsMatchUniqueLemma$ to prove that the fields are equal. This requires proving some types.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5).4</td>
<td>QED</td>
</tr>
<tr>
<td>(6).1</td>
<td>$LL1RAM.authenticator' \in MACType$</td>
</tr>
<tr>
<td></td>
<td>BY (2) 1 DEF $LL2Refinement$, $LL1UntrustedStorageType$</td>
</tr>
<tr>
<td>(6).2</td>
<td>$ll1NewAuthenticator \in MACType$</td>
</tr>
<tr>
<td></td>
<td>BY (3) 5</td>
</tr>
<tr>
<td>(6).3</td>
<td>$LL2RAM.authenticator' \in MACType$</td>
</tr>
<tr>
<td></td>
<td>BY (2) 1, $LL2SubtypeImplicationLemma$ DEF $LL2SubtypeImplication$</td>
</tr>
<tr>
<td>(6).4</td>
<td>$symmetricKey \in SymmetricKeyType$</td>
</tr>
<tr>
<td></td>
<td>BY (5) 1</td>
</tr>
<tr>
<td>(6).5</td>
<td>$LL2NVRAM.symmetricKey' \in SymmetricKeyType$</td>
</tr>
<tr>
<td></td>
<td>BY (2) 1, $LL2SubtypeImplicationLemma$ DEF $LL2SubtypeImplication$</td>
</tr>
<tr>
<td>(6).6</td>
<td>$LL2NVRAM.hashBarrier' \in HashType$</td>
</tr>
<tr>
<td></td>
<td>BY (2) 1, $LL2SubtypeImplicationLemma$ DEF $LL2SubtypeImplication$</td>
</tr>
<tr>
<td>(6).7</td>
<td>QED</td>
</tr>
<tr>
<td></td>
<td>BY (5) 1, (5) 2, (5) 3, (6) 1, (6) 2, (6) 3, (6) 4, (6) 5, (6) 6, $AuthenticatorsMatchUniqueLemma$</td>
</tr>
</tbody>
</table>

The refinement asserts that the RAM record has the appropriate type.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4).5</td>
<td>$LL1RAM' \in LL1UntrustedStorageType$</td>
</tr>
<tr>
<td></td>
<td>BY (2) 1 DEF $LL2Refinement$</td>
</tr>
</tbody>
</table>

We use the $LL1RAMRecordCompositionLemma$ to unify the field equalities into a record equality.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4).6</td>
<td>QED</td>
</tr>
<tr>
<td></td>
<td>BY (4) 1, (4) 2, (4) 3, (4) 4, (4) 5, $LL1RAMRecordCompositionLemma$</td>
</tr>
</tbody>
</table>

The fourth conjunct in $LL1PerformOperation$. This conjunct asserts a record equality, so we prove each field in the record separately.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3) 16</td>
<td>$LL1NVRAM' = [ $</td>
</tr>
<tr>
<td></td>
<td>$historySummary \mapsto ll1NewHistorySummary$,</td>
</tr>
<tr>
<td></td>
<td>$symmetricKey \mapsto LL1NVRAM.symmetricKey]$</td>
</tr>
</tbody>
</table>

The history summary field in the primed Memoir-Basic $NVRAM$ equals the new history summary defined in the $LL1PerformOperation$ action, because the logical history summary in the Memoir-Opt $NVRAM$ and $SPCR$ matches the history summary in the Memoir-Basic $NVRAM$ by refinement. We use the $HistorySummariesMatchUniqueLemma$ to prove the equality.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4).1</td>
<td>$LL1NVRAM.historySummary' = ll1NewHistorySummary$</td>
</tr>
<tr>
<td>(5) 1</td>
<td>HistorySummariesMatch(</td>
</tr>
<tr>
<td></td>
<td>$LL1NVRAM.historySummary', $</td>
</tr>
<tr>
<td></td>
<td>$LL2NVRAMLogicalHistorySummary'$,</td>
</tr>
<tr>
<td></td>
<td>$LL2NVRAM.hashBarrier'$)</td>
</tr>
<tr>
<td></td>
<td>BY (2) 1 DEF $LL2Refinement$</td>
</tr>
</tbody>
</table>

In the Memoir-Opt spec, the logical history summary equals the new history summary defined in the $LL2PerformOperation$ action. Proving this is slightly involved, because we have to show that the updates performed by $LL2PerformOperation$ are logically equivalent to the operation of the $Successor$ operator.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5).2</td>
<td>$LL2NVRAMLogicalHistorySummary' = ll2NewHistorySummary$</td>
</tr>
<tr>
<td>(6) 1</td>
<td>$LL2NVRAMLogicalHistorySummary' = [ $</td>
</tr>
<tr>
<td></td>
<td>$anchor \mapsto LL2NVRAM.historySummaryAnchor'$,</td>
</tr>
<tr>
<td></td>
<td>$extension \mapsto LL2SPCR'$]</td>
</tr>
<tr>
<td>(7) 1</td>
<td>$LL2NVRAM.extensionInProgress' = TRUE$</td>
</tr>
<tr>
<td></td>
<td>BY (3) 2</td>
</tr>
<tr>
<td>(7) 2</td>
<td>$LL2SPCR' \neq BaseHashValue$</td>
</tr>
</tbody>
</table>
We use the `HistorySummariesMatchUniqueLemma` to prove that the fields are equal. This requires proving some types.

\[ (8) 1. \quad \text{LL2SPCR'} = \ll2NewHistorySummary延伸 \]
\[ \text{by (3)2 def } \ll2NewHistorySummary, \ll2CurrentHistorySummary \]
\[ (8) 2. \quad \ll2NewHistorySummary延伸 \neq \text{BaseHashValue} \]
\[ (9) 1. \quad \ll2CurrentHistorySummary \subseteq \text{HistorySummaryType} \]
\[ \text{by (3)3} \]
\[ (9) 2. \quad \text{input} \in \text{InputType} \]
\[ (10) 1. \quad \text{input} \in \ll2AvailableInputs \]
\[ \text{by (3)2} \]
\[ (10) 2. \quad \ll2AvailableInputs \subseteq \text{InputType} \]
\[ \text{by (2)1 def } \ll2TypeInvariant \]
\[ (10) 3. \quad \text{QED} \]
\[ \text{by (10)1, (10)2} \]
\[ (9) 3. \quad \text{LL2NVRAM.hashBarrier} \in \text{HashType} \]
\[ \text{by (2)1, \text{LL2SubtypeImplicationLemma} def } \ll2SubtypeImplication \]
\[ (9) 4. \quad \text{QED} \]
\[ \text{by (9)1, (9)2, (9)3, \text{SuccessorHasNonBaseExtensionLemma}} \]
\[ \text{def } \ll2NewHistorySummary \]
\[ (8) 3. \quad \text{QED} \]
\[ \text{by (8)1, (8)2} \]
\[ (7) 3. \quad \text{QED} \]
\[ \text{by (7)1, (7)2 def } \ll2NVRAMLogicalHistorySummary \]
\[ (6) 2. \quad \ll2NewHistorySummary = [\]
\[ \text{anchor} \mapsto \ll2NVRAM.historySummaryAnchor', \]
\[ \text{extension} \mapsto \ll2SPCR'] \]
\[ (7) 1. \quad \ll2NewHistorySummary.anchor = \ll2NVRAM.historySummaryAnchor' \]
\[ (8) 1. \quad \ll2NewHistorySummary = \]
\[ \text{Successor}(\ll2CurrentHistorySummary, \text{input}, \ll2NVRAM.hashBarrier) \]
\[ \text{by def } \ll2NewHistorySummary \]
\[ (8) 2. \quad \ll2NewHistorySummary.anchor = \ll2CurrentHistorySummary.anchor \]
\[ \text{by (8)1 def } \text{Successor} \]
\[ (8) 3. \quad \ll2NewHistorySummary.anchor = \ll2NVRAM.historySummaryAnchor \]
\[ \text{by (8)2 def } \ll2CurrentHistorySummary \]
\[ (8) 4. \quad \text{UNCHANGED } \ll2NVRAM.historySummaryAnchor \]
\[ \text{by (3)2} \]
\[ (8) 5. \quad \text{QED} \]
\[ \text{by (3)3, (8)4} \]
\[ (7) 2. \quad \ll2NewHistorySummary.extension = \ll2SPCR' \]
\[ \text{by (3)2 def } \ll2NewHistorySummary, \ll2CurrentHistorySummary \]
\[ (7) 3. \quad \ll2NewHistorySummary \in \text{HistorySummaryType} \]
\[ \text{by (3)3} \]
\[ (7) 4. \quad \text{QED} \]
\[ \text{by (7)1, (7)2, (7)3, HistorySummaryRecordCompositionLemma} \]
\[ (6) 3. \quad \text{QED} \]
\[ \text{by (6)1, (6)2} \]
\[ (5) 3. \quad \text{HistorySummariesMatch} (\]
\[ \ll1NewHistorySummary, \ll2NewHistorySummary, \ll2NVRAM.hashBarrier') \]
\[ \text{by (3)10} \]
The primed set of observed authenticators matches across the specs. This follows directly from the refinement.

(4) 1. AuthenticatorSetsMatch(
    LL1ObservedAuthenticators',
    LL2ObservedAuthenticators',
    LL2NVRAM.symmetricKey',
    LL2NVRAM.hashBarrier')

   BY (2) 1 DEF LL2Refinement

   The union matches across the specs. We prove this by proving the matching of each constituent set.

   (4) 2. AuthenticatorSetsMatch(

   ...
\(LL1\text{ObservedAuthenticators} \cup \{ll1\text{NewAuthenticator}\},\)
\(LL2\text{ObservedAuthenticators} \cup \{ll2\text{NewAuthenticator}\},\)
\(LL2\text{NVRAM}\_\text{symmetricKey}',\)
\(LL2\text{NVRAM}\_\text{hashBarrier}'\)

(5.1. UNCHANGED \(\text{LL2NVRAM}\_\text{symmetricKey}, \text{LL2NVRAM}\_\text{hashBarrier}\))

BY \(\langle 3\rangle 2\)

The unprimed set of observed authenticators matches across the specs. This follows directly from the refinement.

(5.2. AuthenticatorSetsMatch(\(\)
    \(LL1\text{ObservedAuthenticators},\)
    \(LL2\text{ObservedAuthenticators},\)
    \(LL2\text{NVRAM}\_\text{symmetricKey}',\)
    \(LL2\text{NVRAM}\_\text{hashBarrier}'\))

BY \(\langle 2\rangle 1, (5) 1\) DEF \text{LL2Refinement}

The new authenticator matches across the specs. This follows directly from the refinement.

(5.3. AuthenticatorsMatch(\(\)
    \(ll1\text{NewAuthenticator},\)
    \(ll2\text{NewAuthenticator},\)
    \(LL2\text{NVRAM}\_\text{symmetricKey}',\)
    \(LL2\text{NVRAM}\_\text{hashBarrier}'\))

BY \(\langle 2\rangle 1, (5) 1\) DEF \text{LL2Refinement}

(5.4. QED)

BY \(\langle 5\rangle 2, (5) 3\) DEF AuthenticatorSetsMatch

In the Memoir-Opt spec, the primed set of observed authenticators is formed from the union of the unprimed set of observed authenticators and the new authenticator.

(4.3. \(LL2\text{ObservedAuthenticators}' =\)
    \(LL2\text{ObservedAuthenticators} \cup \{ll2\text{NewAuthenticator}\}\)

BY \(\langle 3\rangle 2\)

DEF \(ll2\text{NewAuthenticator}, ll2\text{NewHistoryStateBinding}, ll2\text{NewHistorySummaryHash},\)
    \(ll2\text{NewStateHash}, ll2\text{NewPrivateKeyEnc}, ll2\text{SResult}, ll2\text{PrivateKey},\)
    \(ll2\text{NewHistorySummary}, ll2\text{CurrentHistorySummary}\)

(4.4. QED)

We use the AuthenticatorSetsMatchUniqueLemma to prove that the sets are equal. This requires proving some types.

(5.1. \(LL1\text{ObservedAuthenticators}' \in \text{SUBSET MACType}\)

BY \(\langle 2\rangle 1\) DEF \text{LL2Refinement}, \text{LL1UntrustedStorageType}

(5.2. \(LL1\text{ObservedAuthenticators} \cup \{ll1\text{NewAuthenticator}\} \in \text{SUBSET MACType}\)

BY \(\langle 2\rangle 1\) DEF \text{LL2Refinement}, \text{LL1UntrustedStorageType}

(6.1. \(LL1\text{ObservedAuthenticators} \in \text{SUBSET MACType}\)

BY \(\langle 2\rangle 1\) DEF \text{LL2Refinement}, \text{LL1UntrustedStorageType}

(6.2. \(ll1\text{NewAuthenticator} \in \text{MACType}\)

BY \(\langle 3\rangle 5\)

(6.3. QED)

BY \(\langle 6\rangle 1, (6) 2\)

(5.3. \(LL2\text{ObservedAuthenticators}' \in \text{SUBSET MACType}\)

BY \(\langle 2\rangle 1\) DEF \text{LL2TypeInvariant}

(5.4. \(LL2\text{NVRAM}\_\text{symmetricKey}' \in \text{SymmetricKeyType}\)

BY \(\langle 2\rangle 1\), \text{LL2SubtypeImplicationLemma} DEF \text{LL2SubtypeImplication}

(5.5. \(LL2\text{NVRAM}\_\text{hashBarrier}' \in \text{HashType}\)

BY \(\langle 2\rangle 1\), \text{LL2SubtypeImplicationLemma} DEF \text{LL2SubtypeImplication}

(5.6. QED)

BY \(\langle 4\rangle 1, (4) 2, (4) 3, (5) 1, (5) 2, (5) 3, (5) 4, (5) 5\),
We prove that the definitions from \( \text{LL} \) satisfy their types, using the \( \text{LL} \) definitions from \( \text{LL} \).

\section{A Memoir-Opt \( \text{LL} \) action refinement to a Memoir-Basic \( \text{LL} \) action.}

\begin{enumerate}
\item \( \text{LL} \) \( \Rightarrow \) \( \text{LL} \)
\end{enumerate}

We assume the antecedent.

\begin{enumerate}
\item We pick an input that satisfies the \( \text{LL} \) predicate.
\item \( \text{LL} \) \( \Rightarrow \) \( \text{LL} \)
\end{enumerate}

We re-state the definitions from \( \text{LL} \).

\begin{enumerate}
\item \( \text{LL} \) \( \Rightarrow \) \( \text{LL} \)
\end{enumerate}

We prove that the definitions from \( \text{LL} \) satisfy their types, using the \( \text{LL} \) definitions from \( \text{LL} \).
LL2CheckpointedNewHistorySummary.anchor ∈ HashType
LL2CheckpointedNewHistorySummary.extension ∈ HashType
LL2CheckpointedNewCheckpointedHistorySummary ∈ HistorySummaryType
LL2CheckpointedNewCheckpointedHistorySummary.anchor ∈ HashType
LL2CheckpointedNewCheckpointedHistorySummary.extension ∈ HashType
ll2PrivateState ∈ PrivateStateType
ll2SResult ∈ ServiceResultType
ll2SResult.newPublicState ∈ PublicStateType
ll2SResult.newPrivateState ∈ PrivateStateType
ll2SResult.output ∈ OutputType
ll2NewPrivateStateEnc ∈ PrivateStateEncType
ll2CurrentHistorySummary ∈ HistorySummaryType
ll2CurrentHistorySummary.anchor ∈ HashType
ll2CurrentHistorySummary.extension ∈ HashType
ll2CurrentHistorySummaryHash ∈ HashType
ll2NewStateHash ∈ HashType
ll2NewHistoryStateBinding ∈ HashType
ll2NewAuthenticator ∈ MACType

(4).1. input ∈ LL2AvailableInputs
   BY (3) 2
(4).2. LL2TypeInvariant
   BY (2) 1
(4).3. QED
   BY (4).1, (4).2, LL2RepeatOperationDefsTypeSafeLemma

We hide the definitions.

(3) HIDE DEF ll2HistorySummaryHash, ll2HistoryState, ll2HistoryStateBinding, ll2NewHistorySummary,
ll2CheckpointedHistorySummary, ll2NewCheckpointedHistorySummary,
ll2CheckpointedNewHistorySummary, ll2CheckpointedNewCheckpointedHistorySummary,
ll2PrivateState, ll2SResult, ll2NewPrivateStateEnc, ll2CurrentHistorySummary,
ll2CurrentHistorySummaryHash, ll2NewStateHash, ll2NewHistoryStateBinding,
ll2NewAuthenticator

One fact that will be needed many places is that the input is in the Memoir-Basic set of available inputs.

(4).4. input ∈ LL1AvailableInputs
   (4).1. input ∈ LL2AvailableInputs
      BY (3) 2
   (4).2. LL1AvailableInputs = LL2AvailableInputs
      BY (2) 1 DEF LL1Refinement
   (4).3. QED
      BY (4).1, (4).2

We re-state the definitions from LL1RepeatOperation.

(3) ll1StateHash ≜ Hash(L11RAM.publicState, L11RAM.privateStateEnc)
(3) ll1HistoryStateBinding ≜ Hash(L11RAM.historySummary, ll1StateHash)
(3) ll1PrivateState ≜ SymmetricDecrypt(L11NVRAM.symmetricKey, L11RAM.privateStateEnc)
(3) ll1sResult ≜ Service(L11RAM.publicState, ll1PrivateState, input)
(3) ll1NewPrivateStateEnc ≜ SymmetricEncrypt(L11NVRAM.symmetricKey, ll1sResult.newPrivateState)
(3) ll1NewStateHash ≜ Hash(ll1sResult.newPublicState, ll1NewPrivateStateEnc)
(3) ll1NewHistoryStateBinding ≜ Hash(L11NVRAM.historySummary, ll1NewStateHash)
(3) ll1NewAuthenticator ≜ GenerateMAC(L11NVRAM.symmetricKey, ll1NewHistoryStateBinding)

We prove that the definitions from LL1RepeatOperation satisfy their types, using the LL1RepeatOperationDefsTypeSafeLemma and the TypeSafetyRefinementLemma.
The new encrypted private states are directly equal across the two specs.

The service results are directly equal across the two specs.

The private states are directly equal across the two specs.

We prove the correspondences between the definitions in $LL_1$ and $LL_2$. The state hashes are directly equal.

We hide the definitions from $LL_1$.

We prove the correspondences between the definitions in $LL_1$ and $LL_2$. The state hashes are directly equal.

The private states are directly equal across the two specs.

The service results are directly equal across the two specs.

The new encrypted private states are directly equal across the two specs.
There is no definition in the Memoir-Basic spec that corresponds to $ll_2CurrentHistorySummary$ in the Memoir-Opt spec. Instead, the corresponding value is the history summary field of the $NVRAM$ variable.

The history summary in the Memoir-Basic $NVRAM$ matches the logical history summary in the Memoir-Opt $NVRAM$ and $SPCR$, as specified by the refinement.

In the Memoir-Opt spec, the logical history summary in the $NVRAM$ and $SPCR$ equals the $ll_2CurrentHistorySummary$ defined by the $LL_2RepeatOperation$ action. This is a fairly straightforward equivalence by definition, but it requires separate consideration for the two cases of whether an extension is in progress.

In the Memoir-Opt spec, the logical history summary in the $NVRAM$ and $SPCR$ equals the $ll_2CurrentHistorySummary$ defined by the $LL_2RepeatOperation$ action. This is a fairly straightforward equivalence by definition, but it requires separate consideration for the two cases of whether an extension is in progress.
The new authenticators match across the two specs, as specified by the AuthenticatorsMatch predicate.

The new state hashes are directly equal across the two specs.

We then prove that, in the Memoir-Basic spec, the new authenticator is generated as a MAC for the new history state binding. We will use the MACComplete property.

We then prove that, in the Memoir-Opt spec, the new authenticator is a valid MAC for the new history state binding. To do this, we first need to prove some types.

We can thus use the MACComplete property to show that the generated MAC validates appropriately. To do this, we first need to prove some types.

Then, we appeal to the MACComplete property in a straightforward way.

We then prove that, in the Memoir-Basic spec, the new authenticator is generated as a MAC of the new history state binding.

We then prove that, in the Memoir-Opt spec, the new authenticator is generated as a MAC of the new history state binding.
The new history summaries match across the two specs, as we proved above.

We then invoke the definition of the AuthenticatorsMatch predicate.

The first conjunct in LL1RepeatOperation. This is basically just an application of AuthenticatorValidatedLemma.

We need the fact that the symmetric keys in the NVRAM are equal across the two specs.

We prove the types that are needed for AuthenticatorValidatedLemma.

The second precondition also follows from the refinement.

The third precondition follows from the definition of the LL2RepeatOperation action.
By (3), (4), (4.1), (42), (4.3), (44), (45), (46), (47), (48), (49), (410), (411),
\textit{AuthenticatorValidatedLemma}
\textsc{def} \langle l1\text{HistoryStateBinding}, l2\text{HistoryStateBinding}, l2\text{HistorySummaryHash} \rangle

However, the prover seems to get a little confused in this instance. We make life easier for the prover by explicitly staging the instantiation of the quantified variables within the definition of \textit{AuthenticatorValidatedLemma}.

\begin{align*}
\forall \text{samLL2Authenticator} \in \text{MACType}, \\
\text{samSymmetricKey2} \in \text{SymmetricKeyType}, \\
\text{samHashBarrier} \in \text{HashType} : \\
\text{AuthenticatorValidatedLemma}(!l2\text{HashDomain}, LL1\text{RAM}.\text{historySummary}, \\
LL2\text{RAM}.\text{historySummary}, LL1\text{RAM}.\text{authenticator}, \text{samLL2Authenticator}, \\
\text{symmetricKey}, \text{samSymmetricKey2}, \text{samHashBarrier})!1
\end{align*}

By (4)2, (4)3, (4)4, (4)5, \textit{AuthenticatorValidatedLemma}
\begin{align*}
\forall \text{ll2CurrentHistorySummary} = \text{ll2CheckpointedNewHistorySummary} \\
\lor \text{ll2CurrentHistorySummary} = \text{ll2CheckpointedNewCheckpointedHistorySummary}
\end{align*}

By (3)2
\textsc{def} \langle ll2\text{CurrentHistorySummary}, ll2\text{NewHistorySummary}, \\
ll2\text{CheckpointedHistorySummary}, ll2\text{CheckpointedNewHistorySummary}, \\
l2\text{CheckpointedNewCheckpointedHistorySummary} \rangle

Before proceeding to the cases, we'll prove one type fact that will be needed several places below.
\begin{align*}
\text{Hash}(LL1\text{RAM}.\text{historySummary}, \text{input}) \in \text{HashType}
\end{align*}

By (3)1
\textsc{def} \langle ll2\text{Refinement}, LL1\text{UntrustedStorageType} \rangle
\textsc{def} \langle ll1\text{HistoryStateBinding}, ll2\text{HistoryStateBinding}, ll2\text{HistorySummaryHash} \rangle

The second conjunct in \textit{LL1RepeatOperation}: In the Memoir-Basic spec, the history summary in the NVRAM equals the hash of the history summary in the RAM and the input.
(4) 3. \textbf{Case} $\text{ll2CurrentHistorySummary} = \text{ll2CheckpointedNewHistorySummary}$

We will be employing the \textit{HistorySummariesMatchAcrossCheckpointLemma}, which has three preconditions. The first is that the history summaries in the NVRAM match across the two specs, which is true by refinement.

(5) 1. \textit{HistorySummariesMatch}(
   
   $L1\text{NVRAM}\_\text{historySummary}$,
   $L2\text{NVRAMLogicalHistorySummary}$,
   $L2\text{NVRAM}\_\text{hashBarrier}$)

by (2) 1 \textit{def} \textit{LL2Refinement}

The second precondition is that the Memoir-Basic history summary formed from the hash of the history summary in the RAM and the input matches the new history summary in the Memoir-Opt spec.

(5) 2. \textit{HistorySummariesMatch}(
   
   $\text{Hash}(L1\text{RAM}\_\text{historySummary}, \text{input})$,
   $\text{ll2NewHistorySummary}$,
   $L2\text{NVRAM}\_\text{hashBarrier}$)

First, we prove that the \textit{HistorySummariesMatch} predicate equals the \textit{HistorySummariesMatchRecursion} predicate in this case. We assert each condition required by the definition of the predicate.

(6) 1. \textit{HistorySummariesMatch}(
   
   $\text{Hash}(L1\text{RAM}\_\text{historySummary}, \text{input})$,
   $\text{ll2NewHistorySummary}$,
   $L2\text{NVRAM}\_\text{hashBarrier}$) = \textit{HistorySummariesMatchRecursion}(
   
   $\text{Hash}(L1\text{RAM}\_\text{historySummary}, \text{input})$,
   $\text{ll2NewHistorySummary}$,
   $L2\text{NVRAM}\_\text{hashBarrier}$)

We begin by proving the types for the \textit{HistorySummariesMatchRecursion} predicate.

(7) 1. $\text{Hash}(L1\text{RAM}\_\text{historySummary}, \text{input}) \in \text{HashType}$
   
   by (4) 2

(7) 2. $\text{ll2NewHistorySummary} \in \text{HistorySummaryType}$
   
   by (3) 3

(7) 3. $L2\text{NVRAM}\_\text{hashBarrier} \in \text{HashType}$
   
   by (2) 1, $L2\text{SubtypeImplicationLemma}$ \textit{def} $L2\text{SubtypeImplication}$

(7) $\text{ll2InitialHistorySummary} \triangleq [\text{anchor} \rightarrow \text{BaseHashValue}, \text{extension} \rightarrow \text{BaseHashValue}]$

We then prove that this is not the base case for the \textit{HistorySummariesMatch} predicate.

(7) 4. $\text{ll2NewHistorySummary} \neq \text{ll2InitialHistorySummary}$

This proof has a lot of sub-steps, but it is pretty simple. We just use the \textit{BaseHashValueUnique} property to show that the extension field in the \textit{ll2NewHistorySummary} record cannot match the base hash value, which is the value of the extension field in the initial history summary.

(8) 1. $\text{ll2NewHistorySummary} =$ \textit{Successor}($L1\text{RAM}\_\text{historySummary}, \text{input}, L2\text{NVRAM}\_\text{hashBarrier}$)
   
   by (8) 1 \textit{def} \textit{Successor}

(8) 2. $\text{ll2NewHistorySummary}.\text{extension} =$ \textit{Hash}($L2\text{RAM}\_\text{historySummary}.\text{extension}, \text{Hash}(L2\text{NVRAM}\_\text{hashBarrier}, \text{input})$)
   
   by (8) 1 \textit{def} \textit{Successor}

(8) 3. $\text{ll2NewHistorySummary}.\text{extension} \neq \text{BaseHashValue}$

(9) 1. $L2\text{RAM}\_\text{historySummary}.\text{extension} \in \text{HashDomain}$
   
   by (11) 1 \textit{def} \textit{HistorySummaryType}

(10) 1. $L2\text{RAM}\_\text{historySummary}.\text{extension} \in \text{HashType}$
   
   by (2) 1, $L2\text{SubtypeImplicationLemma}$ \textit{def} $L2\text{SubtypeImplication}$

(10) 2. \textbf{q.e.d.}

317
BY (10) 1 DEF HashDomain
(9) 2. Hash(LL2NVRAM.hashBarrier, input) ∈ HashDomain
(10) 1. Hash(LL2NVRAM.hashBarrier, input) ∈ HashType
(11) 1. LL2NVRAM.hashBarrier ∈ HashDomain
(12) 1. LL2NVRAM.hashBarrier ∈ HashType
    BY (2) 1, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication
(12) 2. QED
    BY (12) 1 DEF HashDomain
(11) 2. input ∈ HashDomain
    (12) 1. input ∈ InputType
        (13) 1. input ∈ LL2AvailableInputs
            BY (3) 2
        (13) 2. LL2AvailableInputs ⊆ InputType
            BY (2) 1 DEF LL2TypeInvariant
    (13) 3. QED
        BY (13) 1, (13) 2
    (12) 2. QED
        BY (12) 1 DEF HashDomain
(11) 3. QED
        BY (11) 1, (11) 2, HashTypeSafe
(10) 2. QED
        BY (10) 1 DEF HashDomain
(9) 3. QED
        BY (8) 2, (9) 1, (9) 2, BaseHashValueUnique
(8) 4. QED
    BY (8) 3
Since this is not the base case, the HistorySummariesMatch predicate equals the HistorySummariesMatchRecursion predicate.

(7) 5. QED
    BY (7) 1, (7) 2, (7) 3, (7) 4, HistorySummariesMatchDefinition
Then, we prove that the HistorySummariesMatchRecursion predicate is satisfied. We assert each condition required by the definition of the predicate.

(6) 2. HistorySummariesMatchRecursion(
    Hash(LL1RAM.historySummary, input),
    ll2NewHistorySummary,
    LL2NVRAM.hashBarrier)
We begin by proving the types for the existentially quantified variables in the HistorySummariesMatchRecursion predicate.

(7) 1. input ∈ InputType
    (8) 1. input ∈ LL2AvailableInputs
        BY (3) 2
    (8) 2. LL2AvailableInputs ⊆ InputType
        BY (2) 1 DEF LL2TypeInvariant
    (8) 3. QED
        BY (8) 1, (8) 2
(7) 2. LL1RAM.historySummary ∈ HashType
    BY (2) 1 DEF LL2Refinement, LL1UntrustedStorageType
(7) 3. LL2RAM.historySummary ∈ HistorySummaryType
    BY (2) 1, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication
We then prove the three conjuncts in the HistorySummariesMatchRecursion predicate. The first conjunct follows directly from the refinement.

(7) 4. HistorySummariesMatch(
given that an extension is not in progress when an action is executed.

RepeatOperation

We use the \textit{HistorySummariesMatchAcrossCheckpointLemma} to prove that the fields are equal. This requires proving some types.
In the second case, a checkpoint was taken before the input was processed.

We will be employing the `HistorySummariesMatch` predicate in this case. We assert each condition required by the definition of the predicate.

First, we prove that the `HistorySummariesMatch` predicate equals the `HistorySummariesMatchRecursion` predicate in this case. We assert each condition required by the definition of the predicate.

We then prove that this is not the base case for the `HistorySummariesMatch` predicate.

This proof has a lot of sub-steps, but it is pretty simple. We just use the `BaseHashValueUnique` property to show that the extension field in the `NewCheckpointedHistorySummary` record cannot match the base hash value, which is the value of the extension field in the initial history summary.
BY ⟨8⟩1 DEF Successor
⟨8⟩3. ll2NewCheckpointedHistorySummary.extension ≠ BaseHashValue
(9).1. ll2CheckpointedHistorySummary.extension ∈ HashDomain
(10).1. ll2CheckpointedHistorySummary.extension ∈ HashType
   (11).1. ll2CheckpointedHistorySummary ∈ HistorySummaryType
   BY ⟨3⟩3
   (11).2. Qed
   BY ⟨11⟩1 DEF HistorySummaryType
(10).2. Qed
BY ⟨10⟩1 DEF HashDomain
(9).2. Hash(ll2NVRAM.hashBarrier, input) ∈ HashDomain
(10).1. Hash(ll2NVRAM.hashBarrier, input) ∈ HashType
   (12).1. ll2NVRAM.hashBarrier ∈ HashDomain
   BY ⟨2⟩1, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication
   ⟨12⟩2. Qed
   BY ⟨12⟩1 DEF HashDomain
   ⟨11⟩2. input ∈ HashDomain
   ⟨12⟩1. input ∈ InputType
   ⟨13⟩1. input ∈ ll2AvailableInputs
   BY ⟨3⟩2
   ⟨13⟩2. ll2AvailableInputs ⊆ InputType
   BY ⟨2⟩1 DEF LL2TypeInvariant
   ⟨13⟩3. Qed
   BY ⟨13⟩1, ⟨13⟩2
   ⟨12⟩2. Qed
   BY ⟨12⟩1 DEF HashDomain
   ⟨11⟩3. Qed
   BY ⟨11⟩1, ⟨11⟩2, HashTypeSafe
   ⟨10⟩2. Qed
BY ⟨10⟩1 DEF HashDomain
⟨9⟩.3. Qed
BY ⟨8⟩2, ⟨9⟩1, ⟨9⟩2, BaseHashValueUnique
⟨8⟩4. Qed
BY ⟨8⟩3

Since this is not the base case, the HistorySummariesMatch predicate equals the
HistorySummariesMatchRecursion predicate.

(7).5. Qed
BY ⟨7⟩1, ⟨7⟩2, ⟨7⟩3, ⟨7⟩4, HistorySummariesMatchDefinition

Then, we prove that the HistorySummariesMatchRecursion predicate is satisfied. We assert each condition
required by the definition of the predicate.

(6).2. HistorySummariesMatchRecursion(
   Hash(ll1RAM.historySummary, input),
   ll2NewCheckpointedHistorySummary,
   ll2NVRAM.hashBarrier)

We begin by proving the types for the existentially quantified variables in the
HistorySummariesMatchRecursion predicate.

(7).1. input ∈ InputType
   ⟨8⟩1. input ∈ ll2AvailableInputs
   BY ⟨3⟩2
   ⟨8⟩2. ll2AvailableInputs ⊆ InputType
   BY ⟨2⟩1 DEF LL2TypeInvariant
We then prove the three conjuncts in the `HistorySummariesMatchRecursion` predicate.

The first conjunct is a recursive instance of the `HistorySummariesMatch` predicate, namely that the history summary in the Memoir-Basic RAM matches the checkpointed history summary in the Memoir-Opt RAM. We have to recursively expand the definition.

(7.4) `HistorySummariesMatch`:

\[ LL1RAM \cdot historySummary \in HashType \]

\[ LL1RAM \cdot historySummary = HistorySummaryType \]

\[ LL2RAM \cdot historySummary \in HistorySummaryType \]

\[ LL2NVRAM \cdot hashBarrier \in HashType \]

From the refinement, we know that the history summary in the Memoir-Basic RAM matches the uncheckpointed history summary in the Memoir-Opt RAM.

(8.1) `HistorySummariesMatch`:

\[ LL1RAM \cdot historySummary = HistorySummaryType \]

\[ LL2RAM \cdot historySummary = HistorySummaryType \]

\[ LL2NVRAM . hashBarrier \in HashType \]

We separately consider the base case and recursive case. We will apply the `HistorySummariesMatchDefinition` in both cases, so we prove the necessary types once up front.

(8.2) `LL1RAM . historySummary \in HashType`

(8.3) `Checkpoint(LL2RAM . historySummary) \in HistorySummaryType`

(8.4) `LL2NVRAM . hashBarrier \in HashType`

(8.5) `ll2InitialHistorySummary \triangleq [anchor \mapsto BaseHashValue, extension \mapsto BaseHashValue]`

The base case is very simple.

(8.6) `case LL2RAM . historySummary = ll2InitialHistorySummary`

\[ LL1RAM . historySummary = BaseHashValue \]

\[ LL2RAM . historySummary = HistorySummaryType \]

\[ LL2NVRAM . hashBarrier \in HashType \]

\[ HistorySummariesMatchDefinition \]

The recursive case is pretty involved.
Then, we'll show that the HistorySummariesMatchRecursion predicate holds.

(9) 2. HistorySummariesMatchRecursion

There is only one sub-step, which is showing that the Memoir-Opt checkpointed history summary is not equal to the initial history summary. This follows naturally from the case we’re in, but it takes a lot of tedious steps to get the prover to recognize this.

(10) 1. \textit{Checkpoint}([LL2RAM.historySummary],
LL2NVRAM.hashBarrier) =
HistorySummariesMatchRecursion(
LL1RAM.historySummary,
\textit{Checkpoint}([LL2RAM.historySummary],
LL2NVRAM.hashBarrier))

(11) 1. \textbf{case} \textit{LL2RAM.historySummary}.extension = \textit{BaseHashValue}
   
   12) 1. \textit{Checkpoint}([LL2RAM.historySummary]) = [LL2RAM.historySummary
   
   \textbf{by} (11) 1 \textbf{DEF} \textit{Checkpoint}
   
   12) 2. \textbf{QED}
   
   \textbf{by} (8) 6, (11) 1, (12) 1

(11) 2. \textbf{case} \textit{LL2RAM.historySummary}.extension \neq \textit{BaseHashValue}

  12) 1. \textit{Checkpoint}([LL2RAM.historySummary]) = [anchor \mapsto \textit{Hash}(
LL2RAM.historySummary.anchor,
LL2RAM.historySummary.extension),

  \textit{extension} \mapsto \textit{BaseHashValue}]

  \textbf{by} (11) 2 \textbf{DEF} \textit{Checkpoint}

  12) 2. \textit{Checkpoint}([LL2RAM.historySummary]).anchor = \textit{Hash}(
LL2RAM.historySummary.anchor, LL2RAM.historySummary.extension)

  \textbf{by} (12) 1

  12) 3. \textit{Checkpoint}([LL2RAM.historySummary]).anchor \neq \textit{BaseHashValue}

  13) 1. LL2RAM.historySummary.anchor \in \textit{HashDomain}

  14) 1. LL2RAM.historySummary.anchor \in \textit{HashType}

  15) 1. LL2RAM.historySummary \in \textit{HistorySummaryType}

   \textbf{by} (2) 1, LL2SubtypeImplicationLemma\textbf{DEF} LL2SubtypeImplication

  15) 2. \textbf{QED}

   \textbf{by} (15) 1 \textbf{DEF} \textit{HistorySummaryType}

  14) 2. \textbf{QED}

   \textbf{by} (14) 1 \textbf{DEF} \textit{HashDomain}

  13) 2. LL2RAM.historySummary.extension \in \textit{HashDomain}

  14) 1. LL2RAM.historySummary.extension \in \textit{HashType}

  15) 1. LL2RAM.historySummary \in \textit{HistorySummaryType}

   \textbf{by} (2) 1, LL2SubtypeImplicationLemma\textbf{DEF} LL2SubtypeImplication

  15) 2. \textbf{QED}

   \textbf{by} (15) 1 \textbf{DEF} \textit{HistorySummaryType}

  14) 2. \textbf{QED}

   \textbf{by} (14) 1 \textbf{DEF} \textit{HashDomain}

  13) 3. \textbf{QED}

   \textbf{by} (12) 2, (13) 1, (13) 2, \textit{BaseHashValueUnique}

  12) 4. \textbf{QED}

   \textbf{by} (12) 2, (12) 3

  11) 3. \textbf{QED}

   \textbf{by} (11) 1, (11) 2

  10) 2. \textbf{QED}

   \textbf{by} (8) 2, (8) 3, (8) 4, (10) 1, HistorySummariesMatchDefinition

Then, we’ll show that the HistorySummariesMatchRecursion predicate holds.
We'll first pick values for the existential variables inside the \textit{HistorySummariesMatchRecursion} predicate that satisfy the predicate for the uncheckpointed history summary in the Memoir-Opt RAM. We know such variables exist, because this history summary matches the history summary in the Memoir-Basic RAM, by the refinement.

\begin{enumerate}
\item \textbf{prevInput} \in \textit{InputType}, \textit{previousLL1HistorySummary} \in \textit{HashType}, \textit{previousLL2HistorySummary} \in \textit{HistorySummaryType}:
\[
\begin{align*}
\text{HistorySummariesMatchRecursion} & (\textit{LL1RAM.historySummary}, \textit{LL2RAM.historySummary}, \textit{LL2NVRAM.hashBarrier})! (\text{prevInput}, \text{previousLL1HistorySummary}, \text{previousLL2HistorySummary}) \\
\end{align*}
\]
\end{enumerate}

We then assert all of the conditions necessary to satisfy the \textit{HistorySummariesMatchRecursion} for the checkpointed history summary in the Memoir-Opt RAM. The first three of these are identical to the conditions that hold for the uncheckpointed history summary.

\begin{enumerate}
\item \textbf{prevInput} \in \textit{InputType}
\item \textbf{HistorySummariesMatch} (\textit{previousLL1HistorySummary}, \textit{previousLL2HistorySummary}, \textit{LL2NVRAM.hashBarrier})
\item \textbf{LL1RAM.historySummary} = \text{Hash}(\text{previousLL1HistorySummary}, \text{prevInput})
\end{enumerate}

The last condition is more involved. We show that the checkpointed history summary is a successor of the previous history summary.

\begin{enumerate}
\item \text{LL2HistorySummaryIsSuccessor} (\text{Checkpoint}(\text{LL2RAM.historySummary}), \text{previousLL2HistorySummary}, \text{prevInput}, \text{LL2NVRAM.hashBarrier})
\end{enumerate}

We note that the uncheckpointed history summary is a successor of the previous history summary.

\begin{enumerate}
\item \text{LL2HistorySummaryIsSuccessor} (\text{LL2RAM.historySummary}, \text{previousLL2HistorySummary}, \text{prevInput}, \text{LL2NVRAM.hashBarrier})
\end{enumerate}

The definition of \textit{LL2HistorySummaryIsSuccessor} tells us that there are two ways that the uncheckpointed history summary could be a successor. We will consider the two cases separately.

\begin{enumerate}
\item $\lor \text{LL2RAM.historySummary} = \text{Successor}(\ldots)$
\end{enumerate}
previousLL2HistorySummary,  
prevInput,  
LL2NVRAM.hashBarrier)  
∨ LL2RAM.historySummary =  
   Checkpoint(  
      Successor(  
         previousLL2HistorySummary,  
         prevInput,  
         LL2NVRAM.hashBarrier))

by (11)1 DEF LL2HistorySummaryIsSuccessor

The first case is trivial. If the uncheckpointed history summary is a successor by virtue of the Successor operator, then an application of the Checkpoint operator yields the checkpointed history summary.

(11)3. CASE LL2RAM.historySummary =  
   Successor(previousLL2HistorySummary, prevInput, LL2NVRAM.hashBarrier)  
(12)1. Checkpoint(LL2RAM.historySummary) =  
   Checkpoint(  
      Successor(  
         previousLL2HistorySummary,  
         prevInput,  
         LL2NVRAM.hashBarrier))

by (11)3
(12)2. QED

by (12)1 DEF LL2HistorySummaryIsSuccessor

The second case is only slightly more involved. If the uncheckpointed history summary is a successor by virtue of the Successor operator and the Checkpoint operator, then a second application of the Checkpoint operator is idempotent.

(11)4. CASE LL2RAM.historySummary =  
   Checkpoint(  
      Successor(  
         previousLL2HistorySummary,  
         prevInput,  
         LL2NVRAM.hashBarrier))

(12)1. Checkpoint(LL2RAM.historySummary) =  
   Checkpoint(Checkpoint(  
      Successor(  
         previousLL2HistorySummary,  
         prevInput,  
         LL2NVRAM.hashBarrier)))

by (11)4
(12)2. Checkpoint(LL2RAM.historySummary) =  
   Checkpoint(  
      Successor(  
         previousLL2HistorySummary,  
         prevInput,  
         LL2NVRAM.hashBarrier))

by (12)1 DEF Checkpoint
(12)3. QED
by (12)2 DEF LL2HistorySummaryIsSuccessor
(11)5. QED
by (11)2, (11)3, (11)4

We have thus shown that the conditions for the HistorySummariesMatchRecursion predicate all hold.

325
(10) 6. QED
   BY (8) 2, (8) 3, (8) 4, (10) 2, (10) 3, (10) 4, (10) 5 DEF HistorySummariesMatchRecursion
Since the HistorySummariesMatch predicate equals the HistorySummariesMatchRecursion predicate,
and since the HistorySummariesMatchRecursion predicate is satisfied, the HistorySummariesMatch
predicate is satisfied.

(9) 3. QED
   BY (9) 1, (9) 2
Both the base case and the recursive case are satisfied.

(8) 7. QED
   BY (8) 5, (8) 6
The second conjunct in the HistorySummariesMatchRecursion predicate is that
Hash(LL1RAM.historySummary, input) is equal to itself. We do not bother writing this.
The third conjunct in the HistorySummariesMatchRecursion predicate is true by definition.

(7) 6. LL2HistorySummaryIsSuccessor(
   ll2NewCheckpointedHistorySummary,
   Checkpoint(LL2RAM.historySummary),
   input,
   LL2NVRAM.hashBarrier)
BY DEF LL2HistorySummaryIsSuccessor, 
ll2NewCheckpointedHistorySummary, ll2CheckpointedHistorySummary
All conjuncts in the HistorySummariesMatchRecursion predicate are satisfied.

(7) 7. QED
   BY (7) 1, (7) 2, (7) 3, (7) 4, (7) 6 DEF HistorySummariesMatchRecursion
Since the HistorySummariesMatch predicate equals the HistorySummariesMatchRecursion predicate, and
the latter predicate is satisfied, the former predicate is satisfied.

(6) 3. QED
   BY (6) 1, (6) 2
The third precondition is that in the Memoir-Opt spec, the logical history summary in the NVRAM and SPCR
equals the checkpointed new checkpointed history summary. This follows from a straightforward expansion of
definitions, given that an extension is not in progress when an LL2RepeatOperation action is executed.

(5) 3. LL2NVRAMLogicalHistorySummary = ll2CheckpointedNewCheckpointedHistorySummary
(6) 1. ¬LL2NVRAM.extensionInProgress
   BY (3) 2
(6) 2. LL2NVRAMLogicalHistorySummary = ll2CurrentHistorySummary
(7) 1. LL2NVRAMLogicalHistorySummary = [
   anchor ↦→ LL2NVRAM.historySummaryAnchor,
   extension ↦→ BaseHashValue]
   BY (6) 1 DEF LL2NVRAMLogicalHistorySummary
(7) 2. ll2CurrentHistorySummary = [
   anchor ↦→ LL2NVRAM.historySummaryAnchor,
   extension ↦→ LL2SPCR]
   BY DEF ll2CurrentHistorySummary
(7) 3. LL2SPCR = BaseHashValue
   BY (3) 2, (6) 1
(7) 4. QED
   BY (7) 1, (7) 2, (7) 3
(6) 3. ll2CurrentHistorySummary = ll2CheckpointedNewCheckpointedHistorySummary
   BY (4) 3
(6) 4. QED
   BY (6) 2, (6) 3
We use the `HistorySummariesMatchAcrossCheckpointLemma` to prove that the fields are equal. This requires proving some types.

5. QED

6.1. LL1NVRAM.historySummary ∈ HashType
BY (2)1 DEF Ll2Refinement, LL1TrustedStorageType
6.2. Hash(LL1RAM.historySummary, input) ∈ HashType
BY (4)2
6.3. LL2NVRAMLdigitalHistorySummary ∈ HistorySummaryType
BY (2)1, LL2NVRAMLdigitalHistorySummaryTypeSafe
6.4. ll2NewCheckpointedHistorySummary ∈ HistorySummaryType
BY (3)3
6.5. LL2NVRAM.hashBarrier ∈ HashType
BY (2)1, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication
6.6. QED
BY (5)1, (5)2, (5)3, (6)1, (6)2, (6)3, (6)4, (6)5, HistorySummariesMatchAcrossCheckpointLemma
DEF ll2CheckpointedNewCheckpointedHistorySummary

The two cases are exhaustive.

4.5. QED
BY (4)1, (4)3, (4)4

The third conjunct in `LL1RepeatOperation`. This conjunct asserts a record equality, so we prove each field in the record separately.

3.15. LL1RAM’ = [publicState ↦ l1sResult.newPublicState,
privateStateEnc ↦ l1NewPrivateStateEnc,
historySummary ↦ LL1NVRAM.historySummary,
authenticator ↦ l1NewAuthenticator]

The public state field in the primed Memoir-Basic RAM equals the public state in the result of the service, because the primed RAM variables match across the two specs.

4.1. LL1RAM.publicState’ = l1sResult.newPublicState
5.1. LL1RAM.publicState’ = LL2RAM.publicState’
BY (2)1 DEF Ll2Refinement
5.2. LL2RAM.publicState’ = ll2SResult.newPublicState
BY (3)2 DEF ll2SResult, ll2PrivateState
5.3. ll1sResult.newPublicState = ll2SResult.newPublicState
BY (3)8
5.4. QED
BY (5)1, (5)2, (5)3

The encrypted private state field in the primed Memoir-Basic RAM equals the encrypted private state in the result of the service, because the primed RAM variables match across the two specs.

4.2. LL1RAM.privateStateEnc’ = l1NewPrivateStateEnc
5.1. LL1RAM.privateStateEnc’ = LL2RAM.privateStateEnc’
BY (2)1 DEF Ll2Refinement
5.2. LL2RAM.privateStateEnc’ = ll2NewPrivateStateEnc
BY (3)2 DEF ll2NewPrivateStateEnc, ll2SResult, ll2PrivateState
5.3. l1NewPrivateStateEnc = ll2NewPrivateStateEnc
BY (3)9
5.4. QED
BY (5)1, (5)2, (5)3
In the Memoir-Basic spec, the history summary field in the primed RAM equals the history summary in the unprimed NVRAM. This is because (1) the history summaries in the RAM variables match across the specs by refinement, (2) the history summary in the primed Memoir-Opt RAM is set equal to the refinement. We use the Authenticator field in the primed Memoir-Basic RAM equals the new authenticator defined in the unprimed NVRAM. In the Memoir-Basic spec, the history summary field in the primed RAM equals the history summary in the Memoir-Opt spec, as we proved above. We can thus use the HistorySummariesMatchUniqueLemma to prove the equality.

(4.3) \( LL1RAM\_historySummary' = LL1NVRAM\_historySummary \)

(5.1) HistorySummariesMatch(
    LL1RAM\_historySummary', LL2RAM\_historySummary', LL2NVRAM\_hashBarrier')
    by (2.1) Def LL2Refinement

(5.2) LL2RAM\_historySummary' = ll2CurrentHistorySummary
    by (3.2) Def ll2CurrentHistorySummary

(5.3) HistorySummariesMatch(
    LL1NVRAM\_historySummary,
    ll2CurrentHistorySummary,
    LL2NVRAM\_hashBarrier')
    by (3.10)

We use the HistorySummariesMatchUniqueLemma to prove that the fields are equal. This requires proving some types.

(5) QED

(6.1) LL1RAM\_historySummary' \in HashType
    by (2.1) Def LL2Refinement, LL1UntrustedStorageType

(6.2) LL1NVRAM\_historySummary \in HashType
    by (2.1) Def LL2Refinement, LL1TrustedStorageType

(6.3) LL2RAM\_historySummary' \in HistorySummaryType
    by (2.1), LL2SubtypeImplicationLemma Def LL2SubtypeImplication

(6.4) LL2NVRAM\_hashBarrier' \in HashType
    by (2.1), LL2SubtypeImplicationLemma Def LL2SubtypeImplication

(6.5) QED
    by (5.1), (5.2), (5.3), (6.1), (6.2), (6.3), (6.4), HistorySummariesMatchUniqueLemma

The authenticator field in the primed Memoir-Basic RAM equals the new authenticator defined in the LL1RepeatOperation action, because the authenticators in the RAM variables match across the specs by refinement. We use the AuthenticatorsMatchUniqueLemma to prove the equality.

(4.4) LL1RAM\_authenticator' = ll1NewAuthenticator

(5.1) Pick symmetricKey \in SymmetricKeyType:
    AuthenticatorsMatch(
        LL1RAM\_authenticator',
        LL2RAM\_authenticator',
        symmetricKey,
        LL2NVRAM\_hashBarrier')
    by (2.1) Def LL2Refinement

(5.2) LL2RAM\_authenticator' = ll2NewAuthenticator
    by (3.2)
    Def ll2NewAuthenticator, ll2NewHistoryStateBinding, ll2CurrentHistorySummaryHash,
    ll2NewStateHash, ll2NewPrivateStateEnc, ll2Result, ll2PrivateState,
    ll2NewHistorySummary, ll2CurrentHistorySummary

(5.3) AuthenticatorsMatch(
    ll1NewAuthenticator,
    ll2NewAuthenticator,
    LL2NVRAM\_symmetricKey',
    LL2NVRAM\_hashBarrier')
    by (3.12)
We use the \textit{AuthenticatorsMatchUniqueLemma} to prove that the fields are equal. This requires proving some types.

\begin{enumerate}
\item[(5)] QED
\item[(6)1.] $\text{LL1RAM}.\text{authenticator}' \in \text{MACType}$
\item[(6)2.] $\text{ll1NewAuthenticator} \in \text{MACType}$
\item[(6)3.] $\text{LL2RAM}.\text{authenticator}' \in \text{MACType}$
\item[(6)4.] $\text{symmetricKey} \in \text{SymmetricKeyType}$
\item[(6)5.] $\text{LL2NVRAM}.\text{symmetricKey}' \in \text{SymmetricKeyType}$
\item[(6)6.] $\text{LL2NVRAM}.\text{hashBarrier}' \in \text{HashType}$
\end{enumerate}

The refinement asserts that the RAM record has the appropriate type.

\begin{enumerate}
\item[(4)] QED
\item[(4)1.] $\text{LL1RAM}' \in \text{LL1UntrustedStorageType}$
\item[(4)2.] $\text{LL2Refinement}$
\end{enumerate}

We use the \textit{LL1RAMRecordCompositionLemma} to unify the field equalities into a record equality.

\begin{enumerate}
\item[(4)3.] $\text{ll1sResult}.\text{output} = \text{ll2SResult}.\text{output}$
\item[(4)4.] $\text{LL2ObservedOutputs}' = \text{LL2ObservedOutputs}$
\item[(4)5.] QED
\end{enumerate}

The fourth conjunct in \textit{LL1RepeatOperation}. The set of observed outputs is equal across the two specs.

\begin{enumerate}
\item[(3)16.] $\text{ll1ObservedOutputs}' = \text{ll1ObservedOutputs} \cup \{\text{ll1sResult}.\text{output}\}$
\item[(4)1.] $\text{LL1ObservedOutputs} = \text{LL2ObservedOutputs}$
\item[(4)2.] $\text{LL1ObservedOutputs}' = \text{LL2ObservedOutputs}'$
\item[(4)3.] $\text{ll1sResult}.\text{output} = \text{ll2SResult}.\text{output}$
\item[(4)4.] $\text{LL2ObservedOutputs}' = \text{LL2ObservedOutputs} \cup \{\text{ll2SResult}.\text{output}\}$
\item[(4)5.] QED
\end{enumerate}

The fifth conjunct in \textit{LL1RepeatOperation}. The NVRAM is unchanged by the \textit{UnchangedNVRAMLemma}.

\begin{enumerate}
\item[(3)17.] \textsc{unchanged} $\text{LL1NVRAM}$
\item[(3)18.] \textsc{unchanged} $\text{LL1Disk}$
\end{enumerate}

The sixth conjunct in \textit{LL1RepeatOperation}. The disk is unchanged by the \textit{UnchangedDiskLemma}.

\begin{enumerate}
\item[(3)19.] \textsc{unchanged} $\text{LL1AvailableInputs}$
\end{enumerate}

The seventh conjunct in \textit{LL1RepeatOperation}. The set of available inputs is unchanged by the \textit{UnchangedAvailableInputsLemma}.

\begin{enumerate}
\item[(3)20.] $\text{ll1ObservedAuthenticators}' = \ldots$
\end{enumerate}
The primed set of observed authenticators matches across the specs. This follows directly from the refinement.

(4.1) \textit{AuthenticatorSetsMatch}(
\texttt{LL1ObservedAuthenticators'},
\texttt{LL2ObservedAuthenticators'},
\texttt{LL2NVRAM.symmetricKey'},
\texttt{LL2NVRAM.hashBarrier'})
\textit{by (2)1 DEF LL2Refinement}

The union matches across the specs. We prove this by proving the matching of each constituent set.

(4.2) \textit{AuthenticatorSetsMatch}(
\texttt{LL1ObservedAuthenticators} ∪ \texttt{ll1NewAuthenticator},
\texttt{LL2ObservedAuthenticators} ∪ \texttt{ll2NewAuthenticator},
\texttt{LL2NVRAM.symmetricKey'},
\texttt{LL2NVRAM.hashBarrier'})
\textit{by (3)2 def ll1NewAuthenticator, ll1NewHistoryStateBinding, ll1CurrentHistorySummaryHash, ll1NewPrivateStateEnc, ll1SResult, ll1PrivateState, ll1NewHistorySummary, ll1CurrentHistorySummary}

The unprimed set of observed authenticators matches across the specs. This follows directly from the refinement.

(5.1) \textit{AuthenticatorSetsMatch}(
\texttt{LL1ObservedAuthenticators},
\texttt{LL2ObservedAuthenticators},
\texttt{LL2NVRAM.symmetricKey'},
\texttt{LL2NVRAM.hashBarrier'})
\textit{by (3)1 def ll2NewAuthenticator, ll2NewHistoryStateBinding, ll2CurrentHistorySummaryHash, ll2NewPrivateStateEnc, ll2SResult, ll2PrivateState, ll2NewHistorySummary, ll2CurrentHistorySummary}

We use the \textit{AuthenticatorSetsMatchUniqueLemma} to prove that the sets are equal. This requires proving some types.

(5.1) \texttt{LL1ObservedAuthenticators'} ∈ \texttt{subset MACType}
\textit{by (2)1 DEF LL2Refinement, LL1UntrustedStorageType}

(5.2) \texttt{LL1ObservedAuthenticators} ∪ \texttt{ll1NewAuthenticator} ∈ \texttt{subset MACType}
\textit{by (3)2 def ll2NewAuthenticator, \texttt{ll1NewHistoryStateBinding, ll1CurrentHistorySummaryHash, ll2NewPrivateStateEnc, ll2SResult, ll2PrivateState, ll2NewHistorySummary, ll2CurrentHistorySummary}}

(5.3) \texttt{ll1NewAuthenticator} ∈ \texttt{MACType}
\textit{by (3)5}

330
(6) 3. QED
   BY (6) 1, (6) 2
(5) 3. \( L L 2 O b s e r v e d A u t h e n t i c a t o r s' \in S U B S E T \ M A C T y p e 
   BY (2) 1 DEF \( L L 2 T y p e I n v a r i a n t 
(5) 4. \( L L 2 N V R A M . s y m m e t r i c K e y' \in S y m m e t r i c K e y T y p e 
   BY (2) 1, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication
(5) 5. \( L L 2 N V R A M . h a s h B a r r i e r' \in H a s h T y p e 
   BY (2) 1, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication
(5) 6. QED
   BY (4) 1, (4) 2, (4) 3, (5) 1, (5) 2, (5) 3, (5) 4, (5) 5,
   AuthenticatorSetsMatchUniqueLemma
(3) 21. QED
   BY (3) 4, (3) 13, (3) 14, (3) 15, (3) 16, (3) 17, (3) 18, (3) 19, (3) 20
DEF LL1RepeatOperation, ll1StateHash, ll1HistoryStateBinding, ll1PrivateKey,
   ll1sResult, ll1NewPrivateKeyEnc, ll1NewStateHash,
   ll1NewHistoryStateBinding, ll1NewAuthenticator

A Memoir-Opt \( L L 2 T a k e C h e c k p o i n t \) action refines to a Memoir-Basic stuttering step. This is not at all obvious, and it is
the essence of why the enhancement from Memoir-Basic to Memoir-Opt spec continues to satisfy the implementation.

(2) 6. \( L L 2 T a k e C h e c k p o i n t \Rightarrow U N C H A N G E D \ L L 1 V a r s 
(3) 1. H A V E \ L L 2 T a k e C h e c k p o i n t
We re-state the definition from \( L L 2 T a k e C h e c k p o i n t \).

(3) newHistorySummaryAnchor \( \triangleq \) Hash\( (L L 2 N V R A M . h i s t o r y S u m m a r y A n c h o r , L L 2 S P C R ) \)
We then hide the definition.

(3) H I D E D E F newHistorySummaryAnchor
Two frequently useful facts are that the symmetric key and the hash barrier in the Memoir-Opt NVRAM are
unchanged. This follows directly from the definition of \( L L 2 T a k e C h e c k p o i n t \).

(3) 2. \( \Lambda U N C H A N G E D \ L L 2 N V R A M . s y m m e t r i c K e y 
\Lambda U N C H A N G E D \ L L 2 N V R A M . h a s h B a r r i e r 
(4) 1. \( L L 2 N V R A M ' = [ 
   \text{historySummaryAnchor} \mapsto \text{newHistorySummaryAnchor}, 
   \text{symmetricKey} \mapsto \text{LL2NVRAM.symmetricKey}, 
   \text{hashBarrier} \mapsto \text{LL2NVRAM.hashBarrier}, 
   \text{extensionInProgress} \mapsto \text{false}]
   BY (3) 1 DEF LL2TakeCheckpoint, newHistorySummaryAnchor
(4) 2. QED
   BY (4) 1 DEF LL2TrustedStorageType
We prove the \( U N C H A N G E D \) status for each Memoir-Basic variable in turn. For \( LL1AvailableInputs, 
LL1ObservedOutputs, LL1ObservedAuthenticators, LL1Disk, \) and \( LL1RAM, \) the \( U N C H A N G E D \) status follows
directly from the lemmas we have proven for this purpose.

(3) 3. \( U N C H A N G E D \ L L 1 A v a i l a b l e I n p u t s 
   BY (2) 1, (3) 1, (3) 2, UnchangedAvailableInputsLemma DEF LL2TakeCheckpoint
(3) 4. \( U N C H A N G E D \ L L 1 O b s e r v e d O u t p u t s 
   BY (2) 1, (3) 1, (3) 2, UnchangedObservedOutputsLemma DEF LL2TakeCheckpoint
(3) 5. \( U N C H A N G E D \ L L 1 O b s e r v e d A u t h e n t i c a t o r s 
   BY (2) 1, (3) 1, (3) 2, UnchangedObservedAuthenticatorsLemma DEF LL2TakeCheckpoint
(3) 6. \( U N C H A N G E D \ L L 1 D i s k 
   BY (2) 1, (3) 1, (3) 2, UnchangedDiskLemma DEF LL2TakeCheckpoint
(3) 7. \( U N C H A N G E D \ L L 1 R A M 
   BY (2) 1, (3) 1, (3) 2, UnchangedRAMLemma DEF LL2TakeCheckpoint
To prove the \( U N C H A N G E D \) status of the Memoir-Basic NVRAM value, we prove each of the two fields separately.

(3) 8. \( U N C H A N G E D \ L L 1 N V R A M 

331
Proving the unchanged status of the history summary in the Memoir-Basic NVRAM is quite involved. The top level is straightforward. We simply prove that both the unprimed and primed history summaries in the Memoir-Basic NVRAM match the primed logical history summary in the Memoir-Opt NVRAM and SPCR. Then, from the HistorySummariesMatchUniqueLemma, we conclude that both unprimed and unprimed history summaries in the Memoir-Basic NVRAM are equal.

(4.2) UNCHANGED LL1NVRAM.historySummary

The primed history summary in the Memoir-Basic NVRAM matches the primed logical history summary in the Memoir-Opt NVRAM and SPCR. This follows directly from the refinement, given that the hash barrier in the Memoir-Opt NVRAM has not changed.

(5.1) HistorySummariesMatch(
    LL1NVRAM.historySummary',
    LL2NVRAMLLogicalHistorySummary',
    LL2NVRAM.hashBarrier)

BY (3), (2)1 DEF LL2Refinement

The unprimed history summary in the Memoir-Basic NVRAM matches the primed logical history summary in the Memoir-Opt NVRAM and SPCR. The proof involves three main steps.

(5.2) HistorySummariesMatch(
    LL1NVRAM.historySummary,
    LL2NVRAMLLogicalHistorySummary',
    LL2NVRAM.hashBarrier)

(6) ll2InitialHistorySummary ≜ [anchor ↦ BaseHashValue, extension ↦ BaseHashValue]

First, we pick a set of variables that satisfy the existentials in the HistorySummariesMatchRecursion operator, for the unprimed state. This is more involved than might seem necessary.

(6.1) PICK input ∈ InputType,
      previousLL1HistorySummary ∈ HashType,
      previousLL2HistorySummary ∈ HistorySummaryType

HistorySummariesMatchRecursion(
    LL1NVRAM.historySummary,
    LL2NVRAMLLogicalHistorySummary',
    LL2NVRAM.hashBarrier)

(6.2) ll2InitialHistorySummary

First, we assert that the unprimed history summaries match across the two specs, which follows from the refinement.

(7.1) HistorySummariesMatch(
    LL1NVRAM.historySummary,
    LL2NVRAMLLogicalHistorySummary,
    LL2NVRAM.hashBarrier)

BY (2)1 DEF LL2Refinement

Then, we prove some types, to satisfy the universal quantifiers in HistorySummariesMatchDefinition.

(7.2) LL1NVRAM.historySummary ∈ HashType

BY (2)1 DEF LL2Refinement, LL1TrustedStorageType

(7.3) LL2NVRAMLLogicalHistorySummary ∈ HistorySummaryType

BY (2)1, LL2NVRAMLLogicalHistorySummaryTypeSafe

(7.4) LL2NVRAM.hashBarrier ∈ HashType

BY (2)1, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication

We prove that the Memoir-Opt logical history summary does not equal the initial history summary. This follows from an enablement condition in the LL2TakeCheckpoint action, namely that LL2SPCR does not equal BaseHashValue.

(7.5) LL2NVRAMLLogicalHistorySummary ≠ ll2InitialHistorySummary

⟨8⟩1. LL2NVRAMLLogicalHistorySummary.extension = LL2SPCR
(9.1) \[ \text{LL2NVRAMLogicalHistorySummary} = [ \]
\[ \quad \text{anchor} \mapsto \text{LL2NVRAM.historySummary.Anchor}, \]
\[ \quad \text{extension} \mapsto \text{LL2SPCR} ] \]
(10.1) \[ \text{LL2NVRAM.extensionInProgress} = \text{true} \]
\[ \text{BY \langle 3 \rangle 1 \text{ DEF LL2TakeCheckpoint} \]
(10.2) \[ \text{LL2SPCR} \neq \text{BaseHashValue} \]
\[ \text{BY \langle 3 \rangle 1, \langle 10 \rangle 1 \text{ DEF LL2TakeCheckpoint} \]
(10.3) QED
\[ \text{BY \langle 10 \rangle 1, \langle 10 \rangle 2 \text{ DEF LL2NVRAMLogicalHistorySummary} \]
(9.2) QED
\[ \text{BY \langle 9 \rangle 1 \]
\[ \langle 8 \rangle 2. \text{LL2SPCR} \neq \text{BaseHashValue} \]
\[ \text{BY \langle 3 \rangle 1 \text{ DEF LL2TakeCheckpoint} \]
\[ \langle 8 \rangle 3. \text{ll2InitialHistorySummary.extension} = \text{BaseHashValue} \]
\[ \text{BY DEF ll2InitialHistorySummary} \]
\[ \langle 8 \rangle 4. \text{QED} \]
\[ \text{BY \langle 8 \rangle 1, \langle 8 \rangle 2, \langle 8 \rangle 3 \]
Finally from HistorySummariesMatchDefinition, we can conclude that the quantified HistorySummariesMatchRecursion predicate is satisfied.

(7.6) QED
\[ \text{BY \langle 7 \rangle 1, \langle 7 \rangle 2, \langle 7 \rangle 3, \langle 7 \rangle 4, \langle 7 \rangle 5, \text{HistorySummariesMatchDefinition} \]
\[ \text{DEF HistorySummariesMatchRecursion} \]

Second, we prove that the quantified HistorySummariesMatchRecursion predicate is satisfied for the unprimed history summary in the Memoir-Basic spec and the primed logical history summary in the Memoir-Opt spec.

(6.2) HistorySummariesMatchRecursion(
\[ \quad \text{LL1NVRAM.historySummary,} \]
\[ \quad \text{LL2NVRAMLogicalHistorySummary}, \]
\[ \quad \text{LL2NVRAM.hashBarrier})!( \]
\[ \quad \text{input,} \]
\[ \quad \text{previousLL1HistorySummary,} \]
\[ \quad \text{previousLL2HistorySummary} \]
\[ \]
The quantified predicate is satisfied for the unprimed variables, because we picked appropriate values for the quantifiers above.

(7.1) HistorySummariesMatchRecursion(
\[ \quad \text{LL1NVRAM.historySummary,} \]
\[ \quad \text{LL2NVRAMLogicalHistorySummary,} \]
\[ \quad \text{LL2NVRAM.hashBarrier})!( \]
\[ \quad \text{input,} \]
\[ \quad \text{previousLL1HistorySummary,} \]
\[ \quad \text{previousLL2HistorySummary} \]
\[ \]
\[ \text{BY \langle 6 \rangle 1 \]
We expand the definition of the quantified predicate into its three constituent conjuncts. The first two conjuncts follow directly from the quantified HistorySummariesMatchRecursion predicate for the unprimed history summaries.

(7.2) HistorySummariesMatch(
\[ \quad \text{previousLL1HistorySummary,} \]
\[ \quad \text{previousLL2HistorySummary,} \]
\[ \quad \text{LL2NVRAM.hashBarrier} \]
\[ \]
\[ \text{BY \langle 7 \rangle 1 \]
(7.3) \[ \text{LL1NVRAM.historySummary} = \text{Hash(previousLL1HistorySummary, input)} \]
\[ \text{BY \langle 7 \rangle 1 \]

333
We expand the definition of the $LLHistorySummaryIsSuccessor$ predicate and ignore the first disjunct, since we know that the history summary has been checkpointed.

(7)4. $LLHistorySummaryIsSuccessor$

\[ LLNVRAMLogicalHistorySummary', previousLL2HistorySummary, \]
\[ input, \]
\[ LL2NVRAM .hashBarrier) \]
\[ ⟨ 8 ⟩ \]
\[ \text{Successor}(previousLL2HistorySummary, input, LL2NVRAM .hashBarrier) \]
\[ ⟨ 8 ⟩ \]
\[ \text{checkpointedSuccessorHistorySummary} \]
\[ \text{successorHistorySummary} \]
\[ \text{checkpointedSuccessorHistorySummary} \]
\[ ⟨ 8 ⟩ \]
\[ LLNVRAMLogicalHistorySummary' = \text{checkpointedSuccessorHistorySummary} \]

First, we prove that the $LLNVRAMLogicalHistorySummary$ equals a record with the fields of new history summary anchor and base hash value. This follows fairly directly from the definitions of $LLNVRAMLogicalHistorySummary$ and $LLTakeCheckpoint$.

(9)1. $LLNVRAMLogicalHistorySummary' = [ \]
\[ anchor \mapsto newHistorySummaryAnchor, \]
\[ extension \mapsto BaseHashValue] \]
\[ ⟨ 10 ⟩ \]
\[ LLNVRAM' = [ \]
\[ historySummaryAnchor \mapsto newHistorySummaryAnchor, \]
\[ symmetricKey \mapsto LL2NVRAM .symmetricKey, \]
\[ hashBarrier \mapsto LL2NVRAM .hashBarrier, \]
\[ extensionInProgress \mapsto FALSE] \]
\[ \text{by } ⟨ 3 ⟩ \text{def } LLTakeCheckpoint, newHistorySummaryAnchor \]
\[ ⟨ 10 ⟩ \]
\[ LLNVRAMLogicalHistorySummary' = [ \]
\[ anchor \mapsto LL2NVRAM .historySummaryAnchor', \]
\[ extension \mapsto BaseHashValue] \]
\[ ⟨ 11 ⟩ \]
\[ LLNVRAM .extensionInProgress' = FALSE \]
\[ \text{by } (10)1 \]
\[ ⟨ 11 ⟩ \]
\[ \text{QED} \]
\[ \text{by } ⟨ 10 ⟩ \text{def } LLNVRAMLogicalHistorySummary \]
\[ ⟨ 10 ⟩ \]
\[ LLNVRAM .historySummaryAnchor' = newHistorySummaryAnchor \]
\[ \text{by } (10)1 \]
\[ ⟨ 10 ⟩ \]
\[ \text{QED} \]
\[ \text{by } ⟨ 10 ⟩ , (10)3 \]

Second, we prove that the $checkpointedSuccessorHistorySummary$ also equals a record with the fields of new history summary anchor and base hash value.

(9)2. $checkpointedSuccessorHistorySummary = [ \]
\[ anchor \mapsto newHistorySummaryAnchor, \]
\[ extension \mapsto BaseHashValue] \]

The main step is proving that the successor history summary equals the logical history summary in the Memoir-Opt NVRAM and SPCR.

(10)1. $successorHistorySummary = LLNVRAMLogicalHistorySummary$

From the $LLHistorySummaryIsSuccessor$ predicate, we know that the logical history summary must either equal the successor or the checkpointed successor.

(11)1. $\lor LLNVRAMLogicalHistorySummary = successorHistorySummary$
\[ \lor LLNVRAMLogicalHistorySummary = checkpointedSuccessorHistorySummary \]
\[ ⟨ 12 ⟩ \]
\[ LLHistorySummaryIsSuccessor( \]
\[ LLNVRAMLogicalHistorySummary', previousLL2HistorySummary, \]
\[ input, \]
\[ LL2NVRAM .hashBarrier) \]
The logical history summary does not equal the checkpointed successor.

(11)2. \( LL2\text{NVRAMLogicalHistorySummary} \neq \text{checkpointedSuccessorHistorySummary} \)

The extension field of the logical history summary does not equal the base hash value. This follows from an enablement condition of the \text{TakeCheckpoint} action.

(12)1. \( LL2\text{NVRAMLogicalHistorySummary}.\text{extension} \neq \text{BaseHashValue} \)

(13)1. \( LL2\text{NVRAMLogicalHistorySummary}.\text{extension} = LL2\text{SPCR} \)

(14)1. \( LL2\text{NVRAMLogicalHistorySummary} = [ \)

\begin{align*}
\text{anchor} & \mapsto LL2\text{NVRAM}.\text{historySummaryAnchor}, \\
\text{extension} & \mapsto LL2\text{SPCR}
\end{align*}

(15)1. \( LL2\text{NVRAM}.\text{extensionInProgress} = \text{TRUE} \)

(15)2. \( LL2\text{SPCR} \neq \text{BaseHashValue} \)

(15)3. \( \text{QED} \)

The extension field of the checkpointed successor does equal the base hash value. This follows from the \text{CheckpointHasBaseExtensionLemma}.

(12)2. \( \text{checkpointedSuccessorHistorySummary}.\text{extension} = \text{BaseHashValue} \)

(13)2. \( LL2\text{SPCR} \neq \text{BaseHashValue} \)

(13)3. \( \text{QED} \)

Since the extension field of the logical history summary does not equal the base hash value, but the extension field of the checkpointed successor does equal the base hash value, it follows that these two records are unequal.

(12)3. \( \text{QED} \)

Given that the successor history summary equals the logical history summary in the Memoir-Opt \text{NVRAM} and \text{SPCR}, we can readily derive the specific field values for this record.
\text{(10)}2. \textit{successorHistorySummary} = [
  \text{anchor} \mapsto \text{LL2NVRAM.historySummaryAnchor},
  \text{extension} \mapsto \text{LL2SPCR}]
\text{(11)}1. \text{LL2NVRAMLogicalHistorySummary} = [
  \text{anchor} \mapsto \text{LL2NVRAM.historySummaryAnchor},
  \text{extension} \mapsto \text{LL2SPCR}]
\text{(12)}1. \text{LL2NVRAM.extensionInProgress} = \text{true}
\text{by \text{(3)}1 def LL2TakeCheckpoint}
\text{(12)}2. \text{LL2SPCR} \neq \text{BaseHashValue}
\text{by \text{(3)}1, \text{(12)}1 def LL2TakeCheckpoint}
\text{(12)}3. \text{QED}
\text{by \text{(12)}1, \text{(12)}2 def LL2NVRAMLogicalHistorySummary}
\text{(11)}2. \text{QED}
\text{by \text{(10)}1, \text{(11)}1}

Now, we merely need to show that the the extension field is not equal to the base hash value, so the
definition of the \textit{Checkpoint} operator yields the same value as the \textit{TakeCheckpoint} action.
\text{(10)}3. \text{QED}
\text{(11)}1. \textit{successorHistorySummary.extension} \neq \text{BaseHashValue}
\text{(12)}1. \textit{successorHistorySummary.extension} = \text{LL2SPCR}
\text{by \text{(10)}2}
\text{(12)}2. \text{LL2SPCR} \neq \text{BaseHashValue}
\text{by \text{(3)}1 def LL2TakeCheckpoint}
\text{(12)}3. \text{QED}
\text{by \text{(12)}1, \text{(12)}2}
\text{(11)}2. \text{QED}
\text{by \text{(10)}2, \text{(11)}1}
\text{def checkpointedSuccessorHistorySummary, Checkpoint, newHistorySummaryAnchor}

Since the \text{LL2NVRAMLogicalHistorySummary} and the \text{checkpointedSuccessorHistorySummary} each
equal the same value, they equal each other.
\text{(9)}3. \text{QED}
\text{by \text{(9)}1, \text{(9)}2}
\text{(8)}2. \text{QED}
\text{by \text{(8)}1}
\text{def LL2HistorySummaryIsSuccessor, checkpointedSuccessorHistorySummary,}
\text{successorHistorySummary}

The predicate is satisfied because each of its conjuncts is satisfied.
\text{(7)}5. \text{QED}
\text{by \text{(7)}2, \text{(7)}3, \text{(7)}4}

Third, we prove that for the unprimed Memoir-Basic history summary and the primed Memoir-
Opt logical history summary, the \textit{HistorySummariesMatch} predicate equals the quantified
\textit{HistorySummariesMatchRecursion} predicate.
\text{(6)}3. \text{HistorySummariesMatch(}
  \text{LL1NVRAM.historySummary,}
  \text{LL2NVRAMLogicalHistorySummary'},
  \text{LL2NVRAM.hashBarrier})
\text{= HistorySummariesMatchRecursion(}
  \text{LL1NVRAM.historySummary,}
  \text{LL2NVRAMLogicalHistorySummary'},
  \text{LL2NVRAM.hashBarrier})

We prove some types, to satisfy the universal quantifiers in \textit{HistorySummariesMatchDefinition}.
\text{(7)}1. \text{LL1NVRAM.historySummary} \in \text{HashType}
We prove that the Memoir-Opt logical history summary does not equal the initial history summary. There are three sub-steps.

First, we prove that the primed value of the anchor field in the Memoir-Opt logical history summary equals the new history summary anchor defined in the `TakeCheckpoint` action. This follows fairly directly from the definitions of the `TakeCheckpoint` action and the `LL2NVRAMLLogicalHistorySummary` operator.

\[9\] 1. \( \text{LL2NVRAM}' = [\]
\[9\] 2. \( \text{anchor} \mapsto \text{LL2NVRAM}_1\text{.historySummaryAnchor}' \),
\[9\] 3. \( \text{extension} \mapsto \text{BaseHashValue} \)

By (9) 1
By (9) 2, (9) 3

Second, we prove that the new history summary anchor is not equal to the base hash value. This follows because the new history summary anchor is generated as a hash by the `TakeCheckpoint` action, and the `BaseHashValueUnique` property tells us that no hash value generated by the `Hash` function can equal the base hash value.

\[8\] 2. \( \text{newHistorySummaryAnchor} \neq \text{BaseHashValue} \)

By (8) 1
By (8) 2, (8) 3

Third, the anchor field of the initial history summary equals the base hash value, by definition.

\[8\] 3. \( \text{ll1InitialHistorySummary}_1\text{.anchor} = \text{BaseHashValue} \)

By def `ll1InitialHistorySummary` (8) 4. QED

By (8) 1, (8) 2, (8) 3
Finally, from \textit{HistorySummariesMatchDefinition}, we can conclude that the \textit{HistorySummariesMatch} predicate equals the quantified \textit{HistorySummariesMatchRecursion} predicate.

(7) 5. QED
   BY (7) 1, (7) 2, (7) 3, (7) 4. \textit{HistorySummariesMatchDefinition}
   DEF ll2InitialHistorySummary

We tie the above steps together by first deriving the unquantified \textit{HistorySummariesMatchRecursion} predicate from the quantified predicate, and then proving the straightforward equality.

(6) 4. QED

(7) 1. \textit{HistorySummariesMatchRecursion}(
   \begin{align*}
   & \text{LL1NVRAM\_historySummary,} \\
   & \text{LL2NVRAM\_LogicalHistorySummary',} \\
   & \text{LL2NVRAM\_hashBarrier}
   \end{align*}
\)
\begin{align*}
& \langle 8 \rangle 1. \land \text{input } \in \text{InputType} \\
& \land \text{previousLL1HistorySummary } \in \text{HashType} \\
& \land \text{previousLL2HistorySummary } \in \text{HistorySummaryType}
\end{align*}
BY (6) 1
\langle 8 \rangle 2. QED
BY (6) 2, (8) 1 DEF \textit{HistorySummariesMatchRecursion}

(7) 2. QED
BY (6) 2, (6) 3, (7) 1

We use the \textit{HistorySummariesMatchUniqueLemma} to conclude that both unprimed and unprimed history summaries in the Memoir-Basic NVRAM are equal.

(5) 3. QED

(6) 1. \text{LL1NVRAM\_historySummary'} \in \text{HashType}
BY (2) 1 DEF \textit{LL2Refinement, LL1TrustedStorageType}
(6) 2. \text{LL1NVRAM\_historySummary } \in \text{HashType}
BY (2) 1 DEF \textit{LL2Refinement, LL1TrustedStorageType}
(6) 3. \text{LL2NVRAM\_LogicalHistorySummary'} \in \text{HistorySummaryType}
BY (2) 1, \text{LL2NVRAM\_LogicalHistorySummaryTypeSafe}
(6) 4. \text{LL2NVRAM\_hashBarrier } \in \text{HashType}
BY (2) 1, \textit{LL2SubtypeImplicationLemma} DEF \textit{LL2SubtypeImplication}
(6) 5. QED
BY (5) 1, (5) 2, (6) 1, (6) 2, (6) 3, (6) 4, \textit{HistorySummariesMatchUniqueLemma}

Proving the \textit{UNCHANGED} status of the symmetric key in the Memoir-Basic NVRAM is straightforward, using the lemma we have proven for this purpose.

(4) 3. \textit{UNCHANGED LL1NVRAM\_symmetricKey}
(5) 1. \textit{UNCHANGED LL2NVRAM\_symmetricKey}
BY (3) 2
(5) 2. QED
BY (2) 1, (5) 1, \textit{UnchangedNVRAMSymmetricKeyLemma}

We can then use the \textit{LL1NVRAMRecordCompositionLemma} directly, once we prove some types.

(4) 8. QED

(5) 1. \text{LL1NVRAM } \in \text{LL1TrustedStorageType}
BY (2) 1 DEF \textit{LL2Refinement}
(5) 2. \text{LL1NVRAM'} \in \text{LL1TrustedStorageType}
BY (2) 1 DEF \textit{LL2Refinement}
(5) 3. QED
BY (4) 2, (4) 3, (5) 1, (5) 2, \textit{LL1NVRAMRecordCompositionLemma}

All of the Memoir-Basic variables are unchanged.

(3) 9. QED
BY (3) 3, (3) 4, (3) 5, (3) 6, (3) 7, (3) 8 DEF \textit{LL1Vars}
A Memoir-Opt LL2Restart action refines to one of two actions in the Memoir-Basic spec. If an extension is in progress at the time the LL2Restart occurs, the loss of the SPCR value caused by the LL2Restart is fatal, so this refines to a LL1RestrictedCorruption action in the Memoir-Basic spec, which in turn refines to an HLDie action in the high-level spec. On the other hand, if an extension is not in progress at the time the LL2Restart occurs, the action refines to an LL1Restart action in the Memoir-Basic spec.

(2.7) \text{LL2Restart} \Rightarrow
\begin{align*}
\text{IF LL2NVRAM.extensionInProgress} \\
\text{THEN} \\
\text{LL1RestrictedCorruption} \\
\text{ELSE} \\
\text{LL1Restart}
\end{align*}

We assume the antecedent.

(3.1) \text{HAVE LL2Restart}

We pick a set of variables of the appropriate types that satisfy the \text{LL2CorruptRAM} action.

(3.2) \text{PICK } ll2UntrustedStorage \in LL2UntrustedStorageType, \\
\quad ll2RandomSymmetricKey \in SymmetricKeyType \setminus \{LL2NVRAM.symmetricKey\}, \\
\quad ll2Hash \in HashType : \\
\quad LL2Restart!(ll2UntrustedStorage, ll2RandomSymmetricKey, ll2Hash)

\text{BY (3.1) DEF LL2Restart}

We first prove that the primed state of the Memoir-Basic RAM has a value that satisfies a particular constraint that is imposed by both the LL1RestrictedCorruption action and by the LL1Restart action.

(3.3) \exists ll1UntrustedStorage \in LL1UntrustedStorageType, \\
\quad ll1RandomSymmetricKey \in SymmetricKeyType \setminus \{LL1NVRAM.symmetricKey\}, \\
\quad ll1Hash \in HashType : \\
\quad ll1UntrustedStorage.authenticator = \\
\quad GenerateMAC(ll1RandomSymmetricKey, ll1Hash) \\
\quad \land LL1RAM’ = ll1UntrustedStorage

We pick a symmetric key that satisfies the AuthenticatorsMatch predicate for the primed states of the authenticators in the RAM variables of the two specs.

(3.4) \text{PICK symmetricKey} \in SymmetricKeyType :
\begin{align*}
\text{AuthenticatorsMatch}(& \\
& LL1RAM.authenticator’, \\
& LL2RAM.authenticator’, \\
& symmetricKey, \\
& LL2NVRAM.hashBarrier’)
\end{align*}

\text{BY (2.1) DEF LL2Refinement}

We pick a set of variables of the appropriate types that satisfy the quantified AuthenticatorsMatch predicate.

(3.4) \text{PICK stateHash} \in HashType, \\
\quad ll1HistorySummary \in HashType, \\
\quad ll2HistorySummary \in HistorySummaryType :
\begin{align*}
\text{AuthenticatorsMatch}(& \\
& ll1HistorySummary, \\
& ll2HistorySummary
\end{align*}

\text{BY (4.3) DEF AuthenticatorsMatch}

We re-state the definitions from the \text{LET} in AuthenticatorsMatch.

(4.1) \text{ll2HistoryStateBinding} \triangleq Hash(ll1HistorySummary, stateHash) \\
(4.2) \text{ll2HistorySummaryHash} \triangleq Hash(ll2HistorySummary.anchor, ll2HistorySummary.extension) \\
(4.3) \text{ll2HistoryStateBinding} \triangleq Hash(ll2HistorySummaryHash, stateHash)
We prove the types of the definitions, with help from the \textit{AuthenticatorsMatchDefsTypeSafeLemma}.

\begin{align*}
(4) &. \wedge ll_1\text{HistoryStateBinding} \in \text{HashType} \\
& \wedge ll_2\text{HistorySummaryHash} \in \text{HashType} \\
& \wedge ll_2\text{HistoryStateBinding} \in \text{HashType} \\
\end{align*}

We hide the definitions.

\begin{align*}
(4) &. \text{HIDE DEF } ll_1\text{HistoryStateBinding}, ll_2\text{HistorySummaryHash}, ll_2\text{HistoryStateBinding} \\
\end{align*}

To prove the constraints regarding the primed state of the Memoir-Basic RAM, we first prove the types of the three witnesses for the existentially quantified variables.

\begin{align*}
(4) &. LL_1\text{RAM}^\prime \in LL_1\text{UntrustedStorageType} \\
\end{align*}

We need to prove some types.

\begin{align*}
(4) &. \text{HIDE DEF } ll_1\text{HistoryStateBinding}, ll_2\text{HistorySummaryHash}, ll_2\text{HistoryStateBinding} \\
\end{align*}

We prove that the authenticator in the Memoir-Basic spec is generated as a MAC with a random symmetric key. This follows directly from the \textit{AuthenticatorGeneratedLemma}, once we prove the preconditions for the lemma.

\begin{align*}
(4) &. LL_1\text{RAM}.\text{authenticator}^\prime = \text{GenerateMAC}(ll_2\text{RandomSymmetricKey}, ll_1\text{HistoryStateBinding}) \\
\end{align*}

Then we prove the three conjuncts in the antecedent of the \textit{AuthenticatorGeneratedLemma}. The first conjunct follows from a conjunct in the refinement.

\begin{align*}
(5) &. \text{stateHash} \in \text{HashType} \\
\end{align*}

\begin{align*}
(5) &. ll_1\text{HistorySummary} \in \text{HashType} \\
\end{align*}

\begin{align*}
(5) &. ll_2\text{HistorySummary} \in \text{HistorySummaryType} \\
\end{align*}

\begin{align*}
(5) &. LL_1\text{RAM}.\text{authenticator}^\prime \in \text{MACType} \\
\end{align*}

\begin{align*}
(5) &. LL_2\text{RAM}.\text{authenticator}^\prime \in \text{MACType} \\
\end{align*}

\begin{align*}
(5) &. \text{HistorySummariesMatch}(ll_1\text{HistorySummary}, ll_2\text{HistorySummary}, LL_2\text{NVRAM}.\text{hashBarrier}^\prime) \\
\end{align*}
The second conjunct in the antecedent of the AuthenticatorGeneratedLemma mainly follows from the refinement, but we also have to prove that the symmetric key specified by an existential in the refinement matches the random symmetric key specified in the LL2CorruptRAM action. This follows from the MACUnforgeable property.

\[ \text{AuthenticatorsMatch} \]

\[
\begin{align*}
\text{LL1RAM}\_\text{authenticator}', \\
\text{LL2RAM}\_\text{authenticator}', \\
\text{ll2RandomSymmetricKey}, \\
\text{LL2NVRAM}\_\text{hashBarrier}'
\end{align*}
\]

The main precondition for the MACUnforgeable property is that the generated MAC is validated. We first prove the validation, which follows from the refinement.

\[ \text{ValidateMAC} \]

\[
\begin{align*}
\text{symmetricKey}, \\
\text{ll2HistoryStateBinding}, \\
\text{LL2RAM}\_\text{authenticator}'
\end{align*}
\]

We then prove the generation, which follows from the LL2CorruptRAM action.

\[ \text{LL2RAM}\_\text{authenticator}' = \text{GenerateMAC} \]

\[
\begin{align*}
\text{ll2RandomSymmetricKey}, \\
\text{ll2Hash}
\end{align*}
\]

The remaining preconditions are types.

\[ \text{symmetricKey} \in \text{SymmetricKeyType} \]

\[
\begin{align*}
\text{ll2RandomSymmetricKey} \in \text{SymmetricKeyType}
\end{align*}
\]

The MACUnforgeable property tells us that the two keys are equal.

\[ \text{MACUnforgeable} \]

\[
\begin{align*}
\text{ll2RandomSymmetricKey} \in \text{SymmetricKeyType}
\end{align*}
\]

The third conjunct in the antecedent of the AuthenticatorGeneratedLemma mainly follows from the LL2CorruptRAM action, but we also have to prove that the history state binding specified in the refinement matches the arbitrary hash specified in the LL2CorruptRAM action.

\[ \text{LL2RAM}\_\text{authenticator}' = \text{GenerateMAC} \]

\[
\begin{align*}
\text{ll2RandomSymmetricKey}, \\
\text{ll2HistoryStateBinding}
\end{align*}
\]

The state authenticator in the primed Memoir-Opt RAM equals the authenticator specified by the existential ll2UntrustedStorage in the LL2CorruptRAM action.
The authenticator specified by the existential ll2UntrustedStorage in the LL2CorruptRAM action is generated as a MAC of the history state binding from the AuthenticatorsMatch predicate invoked by the refinement. This follows from the MACCollisionResistant property.

(6) 2. ll2UntrustedStorage.authoritative =
    GenerateMAC(ll2RandomSymmetricKey, ll2HistoryStateBinding)

The main precondition for the MACUnforgeable property is that the generated MAC is validated. We first prove the validation, which follows from the refinement.

(7) 1. ValidateMAC(symmetricKey, ll2HistoryStateBinding, LL2RAM.authoritative')
    BY (4) 4 DEF ll2HistoryStateBinding, ll2HistorySummaryHash

We then prove the generation, which follows from the LL2CorruptRAM action.

(7) 2. LL2RAM.authoritative' = GenerateMAC(ll2RandomSymmetricKey, ll2Hash)

    ⟨8⟩ 1. LL2RAM' = ll2UntrustedStorage
        BY (3) 2
    ⟨8⟩ 2. ll2UntrustedStorage.authoritative = GenerateMAC(ll2RandomSymmetricKey, ll2Hash)
        BY (3) 2
    ⟨8⟩ 3. QED
        BY (8) 1, (8) 2

The remaining preconditions are types.

(7) 3. symmetricKey ∈ SymmetricKeyType
    BY (4) 3
(7) 4. ll2RandomSymmetricKey ∈ SymmetricKeyType
    ⟨8⟩ 1. ll2RandomSymmetricKey ∈ SymmetricKeyType \ \{LL2NVRAM.symmetricKey\}
        BY (3) 2
    ⟨8⟩ 2. QED
        BY (8) 1
(7) 5. ll2HistoryStateBinding ∈ HashType
    BY (4) 5
(7) 6. ll2Hash ∈ HashType
    BY (3) 2

The MACCollisionResistant property tells us that the two hash values are equal.

(7) 7. ll2Hash = ll2HistoryStateBinding
    BY (7) 1, (7) 2, (7) 3, (7) 4, (7) 5, (7) 6, MACCollisionResistant
(7) 8. QED
    BY (3) 2, (7) 7
(6) 3. QED
    BY (6) 1, (6) 2

We then invoke the AuthenticatorGeneratedLemma directly.

(5) 11. QED
    BY (5) 1, (5) 2, (5) 3, (5) 4, (5) 5, (5) 6, (5) 7, (5) 8, (5) 9, (5) 10,
    AuthenticatorGeneratedLemma
   DEF ll1HistoryStateBinding, ll2HistorySummaryHash, ll2HistoryStateBinding
(4) 10. QED
    BY (4) 6, (4) 7, (4) 8, (4) 9

Since the refinement is an if - then - else , we prove the then and else cases separately. For the then case, we assume that an extension is in progress and show that this refines to a LL1RestrictedCorruption action.

(3) 4. ASSUME LL2NVRAM.extensionInProgress
    PROVE LL1RestrictedCorruption

One fact that will be useful in several places below is that the extension field in the logical history summary of the Memoir-Opt NVRAM and SPCR is equal to a crazy hash value, because an extension is in progress but the SPCR (in the primed state) equals the base hash value.

(4) 1. LL2NVRAMLogicalHistorySummary.extension' = CrazyHashValue

342
We prove each conjunct of $LL1RestrictedCorruption$ separately. First, we prove the conjunct relating to the NVRAM.

(4.2) $LL1RestrictedCorruption!nvram$

The primed value of the history summary field in the Memoir-Basic NVRAM serves as our witness for the garbage history summary.

(5.1) $LL1NVRAM.historySummary' \in HashType$

We prove that the constraint labeled current in the $LL1RestrictedCorruption$ action is satisfied.

(5.2) $LL1RestrictedCorruption!nvram!current(LL1NVRAM.historySummary')$

To prove the universally quantified expression, we take a set of variables of the appropriate types.

(6.1) TAKE stateHash1 \in HashType,

ll1Authenticator \in LL1ObservedAuthenticators

We re-state the definition from within the $LL1RestrictedCorruption!nvram!current$ clause.

(6) ll1GarbageHistoryStateBinding \triangleq Hash(LL1NVRAM.historySummary', stateHash1)

We hide the definition.

(6) HIDE DEF ll1GarbageHistoryStateBinding

We need to prove the $nvram::current$ conjunct, which asserts that the authenticator is not a valid MAC for the history state binding formed from the history summary in the NVRAM and any state hash.

(6.2) \neg ValidateMAC(

LL1NVRAM.symmetricKey,

ll1GarbageHistoryStateBinding,

ll1Authenticator)

We will use proof by contradiction.

(7.1) SUFFICES

ASSUME

ValidateMAC(

LL1NVRAM.symmetricKey,

ll1GarbageHistoryStateBinding,

ll1Authenticator)

PROVE

FALSE

OBVIOUS

We first pick, from the set of Memoir-Opt observed authenticators, a Memoir-Opt authenticator that matches the Memoir-Basic authenticator. We know that such a authenticator exists, because the refinement asserts that the sets of observed authenticators match across the two specs.
(7) 2. Pick \( l2\text{Authenticator} \in LL2\text{ObservedAuthenticators} : \)
 GarnishAuthenticatorsMatch(
  \( ll1\text{Authenticator}, \)
  \( ll2\text{Authenticator}, \)
  \( LL2\text{NVRAM}.\text{symmetricKey}, \)
  \( LL2\text{NVRAM}.\text{hashBarrier} \))

(8) 1. \( ll1\text{Authenticator} \in LL1\text{ObservedAuthenticators} \)

By (6) 1

(8) 2. AuthenticatorSetsMatch(
  \( LL1\text{ObservedAuthenticators}, \)
  \( LL2\text{ObservedAuthenticators}, \)
  \( LL2\text{NVRAM}.\text{symmetricKey}, \)
  \( LL2\text{NVRAM}.\text{hashBarrier} \))

By (2) 1 DEF LL2Refinement

(8) 3. QED

By (8) 1, (8) 2 DEF AuthenticatorSetsMatch

We pick a set of variables of the appropriate types that satisfy the quantified \( \text{AuthenticatorsMatch} \) predicate.

(7) 3. Pick \( \text{stateHash}^2 \in \text{HashType}, \)

\( ll1\text{HistorySummary} \in \text{HashType}, \)

\( ll2\text{HistorySummary} \in \text{HistorySummaryType} : \)

\( \text{AuthenticatorsMatch} (\)
  \( ll1\text{Authenticator}, \)
  \( ll2\text{Authenticator}, \)
  \( LL2\text{NVRAM}.\text{symmetricKey}, \)
  \( LL2\text{NVRAM}.\text{hashBarrier} \) \)

(7) 2 DEF AuthenticatorsMatch

We re-state the definitions from the \text{LET in AuthenticatorsMatch}.

(7) \( ll1\text{HistoryStateBinding} \triangleq \text{Hash}(ll1\text{HistorySummary}, \text{stateHash}^2) \)

(7) \( ll2\text{HistorySummaryHash} \triangleq \text{Hash}(ll2\text{HistorySummary}.\text{anchor}, ll2\text{HistorySummary}.\text{extension}) \)

(7) \( ll2\text{HistoryStateBinding} \triangleq \text{Hash}(ll2\text{HistorySummaryHash}, \text{stateHash}^2) \)

We prove the types of the definitions, with help from the \text{AuthenticatorsMatchDefsTypeSafeLemma}.

(7) 4. \& \( ll1\text{HistoryStateBinding} \in \text{HashType} \)

\( ll2\text{HistorySummaryHash} \in \text{HashType} \)

\( ll2\text{HistoryStateBinding} \in \text{HashType} \)

By (7) 3, AuthenticatorsMatchDefsTypeSafeLemma

We hide the definitions.

(7) \text{HIDE DEF} ll1\text{HistoryStateBinding}, ll2\text{HistorySummaryHash}, ll2\text{HistoryStateBinding}

We prove that the Memoir-Basic s history summary picked to satisfy the \( \text{AuthenticatorsMatch} \) predicate equals the history summary in the primed state of the Memoir-Basic NVRAM.

(7) 5. \( ll1\text{HistorySummary} = LL1NVRAM.\text{historySummary}' \)

The first step is to show the equality of the history state bindings that bind each of these history summaries to their respective state hashes.

(8) 1. \( ll1\text{HistoryStateBinding} = ll1\text{GarbageHistoryStateBinding} \)

By hypothesis, the authenticator is a valid MAC for the garbage history state binding.

(9) 1. \text{ValidateMAC}(
  \( LL2\text{NVRAM}.\text{symmetricKey}, ll1\text{GarbageHistoryStateBinding}, ll1\text{Authenticator} \))

(10) 1. \( LL1NVRAM.\text{symmetricKey} = LL2NVRAM.\text{symmetricKey} \)

By (2) 1 DEF LL2Refinement

(10) 2. QED
The definition of the AuthenticatorsMatch predicate tells us that the Memoir-Basic authenticator was generated as a MAC from the history state binding.

\[ \text{AuthenticatorsMatch} \]

\[ \text{GenerateMAC(LL2NVRAM.symmetricKey, ll1HistoryStateBinding)} \]

The remaining preconditions are types.

\[ \text{Authenticator} = \text{GenerateMAC(LL2NVRAM.symmetricKey, ll1HistoryStateBinding)} \]

The MACCollisionResistant property tells us that the two history state bindings are equal.

\[ \text{MACCollisionResistant} \]

By the collision resistance of the hash function, the equality of the history state bindings implies the equality of the history summaries.

\[ \text{HashCollisionResistant} \]

The Memoir-Basic s history summary picked to satisfy the AuthenticatorsMatch predicate matches the primed logical history summary in the Memoir-Opt NVRAM and SPCR, by the refinement and the above equality.

\[ \text{HistorySummariesMatch(} \]

\[ \text{ll1HistorySummary,} \]

\[ \text{ll1GarbageHistoryStateBinding} \]

\[ \text{ll1HistorySummary} \in \text{HashType} \]

\[ \text{ll1GarbageHistoryStateBinding} \in \text{HashType} \]

\[ \text{ll1HistorySummary} \in \text{HashDomain} \]

\[ \text{ll1GarbageHistoryStateBinding} \in \text{HashDomain} \]

\[ \text{ll1HistorySummary} \in \text{HashType} \]

\[ \text{ll1GarbageHistoryStateBinding} \in \text{HashType} \]

\[ \text{ll1HistorySummary} \in \text{HashDomain} \]

\[ \text{ll1GarbageHistoryStateBinding} \in \text{HashDomain} \]

\[ \text{ll1HistorySummary} \in \text{HashType} \]

\[ \text{ll1GarbageHistoryStateBinding} \in \text{HashType} \]

\[ \text{ll1HistorySummary} \in \text{HashDomain} \]

\[ \text{ll1GarbageHistoryStateBinding} \in \text{HashDomain} \]

\[ \text{ll1HistorySummary} \in \text{HashType} \]

\[ \text{ll1GarbageHistoryStateBinding} \in \text{HashType} \]

\[ \text{ll1HistorySummary} \in \text{HashDomain} \]

\[ \text{ll1GarbageHistoryStateBinding} \in \text{HashDomain} \]

\[ \text{ll1HistorySummary} \in \text{HashType} \]

\[ \text{ll1GarbageHistoryStateBinding} \in \text{HashType} \]
\text{We prove that the } \text{HistorySummariesMatch} \text{ predicate equals the } \text{HistorySummariesMatchRecursion} \text{ predicate in this case. We assert each condition required by the definition of the predicate.}

\text{We prove some types, to satisfy the universal quantifiers in } \text{HistorySummariesMatchDefinition}.\text{ predicate.}

\text{Finally, from } \text{HistorySummariesMatchDefinition}, \text{ we can conclude that the } \text{HistorySummariesMatch} \text{ predicate equals the quantified } \text{HistorySummariesMatchRecursion} \text{ predicate.}

\text{We pick values for the existential variables inside the } \text{HistorySummariesMatchRecursion} \text{ predicate that satisfy the predicate. We know such variables exist, because the predicate is satisfied by the two previous steps.}

\text{We prove some types, to satisfy the universal quantifiers in } \text{HistorySummariesMatchRecursion} \text{ predicate.}
One of the conjuncts in the definition of HistorySummariesMatchRecursion is that the Memoir-Opt history summary is a successor of a previous history summary.

We re-state the definitions from the let in LL2HistorySummaryIsSuccessor.

We re-state a definition from the Succesor operator.

We hide the definitions.

First, we prove that the logical history summary cannot be a successor.

We re-state a definition from the Successor operator.

There is only one sub-step, which is proving that the extension fields of these two records are unequal.

If an extension is in progress but the SPCR equals the base hash value, the logical history summary equals a crazy hash value, as proven above.

The extension field of the successor history summary is equal to a hash generated by the hash function.

The arguments to the hash function are both in the hash domain.
The crazy hash value is not equal to any hash value that can be generated by the hash function when operating on arguments within its domain.

Second, we prove that the logical history summary cannot be a checkpoint.

The extension field of the logical history summary does not equal the base hash value, as we proved above.

The extension field of the checkpointed successor does equal the base hash value. This follows from the CheckpointHasBaseExtensionLemma.
We thus have a contradiction.

(7)13. QED
   BY (7)10, (7)11, (7)12
(6)3. QED
   BY (6)2 DEF ll1GarbageHistoryStateBinding

We prove that the constraint labeled previous in the LL1RestrictedCorruption action is satisfied.

(5)3. LL1RestrictedCorruption!nvram!previous(LL1NVRAM.historySummary')

To prove the universally quantified expression, we take a set of variables of the appropriate types.

(6)1. TAKE stateHash1 ∈ HashType,
    ll1Authenticator ∈ LL1ObservedAuthenticators,
    ll1SomeHistorySummary ∈ HashType,
    someInput ∈ InputType

We re-state the definition from within the LL1RestrictedCorruption!nvram!previous clause.

(6) ll1SomeHistoryStateBinding ≜ Hash(ll1SomeHistorySummary, stateHash1)

We hide the definition.

(6) HIDE DEF ll1SomeHistoryStateBinding

We need to prove the nvram:: previous conjunct, which asserts an implication. It suffices to assume the antecedent and prove the consequent.

(6)2. SUFFICES
    ASSUME LL1NVRAM.historySummary' = Hash(ll1SomeHistorySummary, someInput)
    PROVE ¬ValidateMAC(
        LL1NVRAM.symmetricKey,
        ll1SomeHistoryStateBinding,
        ll1Authenticator)
    BY DEF ll1SomeHistoryStateBinding

The consequent of the nvram:: previous conjunct asserts that the authenticator is not a valid MAC for the history state binding formed from any predecessor of the history summary in the NVRAM and any state hash.

(6)3. ¬ValidateMAC(
    LL1NVRAM.symmetricKey,
    ll1SomeHistoryStateBinding,
    ll1Authenticator)

We will use proof by contradiction.

(7)1. SUFFICES
    ASSUME ValidateMAC(
        LL1NVRAM.symmetricKey,
        ll1SomeHistoryStateBinding,
        ll1Authenticator)
    PROVE FALSE
    OBVIOUS

We first pick, from the set of Memoir-Opt observed authenticators, a Memoir-Opt authenticator that matches the Memoir-Basic authenticator. We know that such a authenticator exists, because the refinement asserts that the sets of observed authenticators match across the two specs.

(7)2. PICK ll2Authenticator ∈ LL2ObservedAuthenticators :
    AuthenticatorsMatch(
        ll1Authenticator,
        ll2Authenticator,
        LL2NVRAM.symmetricKey,
\[ LL2NVRAM.hashBarrier \]

\[ \text{(8.1) } ll1.Authenticator \in LL1.ObservedAuthenticators \]

By (6.1)

\[ \text{(8.2) } \text{AuthenticatorSetsMatch(} \]

\[ LL1.ObservedAuthenticators, \]

\[ LL2.ObservedAuthenticators, \]

\[ LL2NVRAM.symmetricKey, \]

\[ LL2NVRAM.hashBarrier) \]

By (2.1) \text{def } LL2Refinement

\[ \text{(8.3) } \text{QED} \]

By (8.1), (8.2) \text{def } \text{AuthenticatorSetsMatch}

We pick a set of variables of the appropriate types that satisfy the quantified \text{AuthenticatorsMatch} predicate.

\[ \text{(7.3) } \text{pick } stateHash2 \in \text{HashType}, \]

\[ ll1.HistorySummary \in \text{HashType}, \]

\[ ll2.HistorySummary \in \text{HistorySummaryType} : \]

\[ \text{AuthenticatorsMatch(} \]

\[ ll1.Authenticator, \]

\[ ll2.Authenticator, \]

\[ LL2NVRAM.symmetricKey, \]

\[ LL2NVRAM.hashBarrier) \]

\[ \text{!}(stateHash2, ll1.HistorySummary, ll2.HistorySummary) \]

By (7.2) \text{def } \text{AuthenticatorsMatch}

We re-state the definitions from the \text{LET} in \text{AuthenticatorsMatch}.

\[ \text{(7) } ll1.HistoryStateBinding \xrightarrow{\Delta} \text{Hash}(ll1.HistorySummary, stateHash2) \]

\[ \text{(7) } ll2.HistorySummaryHash \xrightarrow{\Delta} \text{Hash}(ll2.HistorySummary.anchor, ll2.HistorySummary.extension) \]

\[ \text{(7) } ll2.HistoryStateBinding \xrightarrow{\Delta} \text{Hash}(ll2.HistorySummaryHash, stateHash2) \]

We prove the types of the definitions, with help from the \text{AuthenticatorsMatchDefsTypeSafeLemma}.

\[ \text{(7.4) } \land \text{ll1.HistoryStateBinding} \in \text{HashType} \]

\[ \land \text{ll2.HistorySummaryHash} \in \text{HashType} \]

\[ \land \text{ll2.HistoryStateBinding} \in \text{HashType} \]

By (7.3), \text{AuthenticatorsMatchDefsTypeSafeLemma}

We hide the definitions.

\[ \text{(7) } \text{hide def ll1.HistoryStateBinding, ll2.HistorySummaryHash, ll2.HistoryStateBinding} \]

We prove that the Memoir-Basic's history summary picked to satisfy the \text{AuthenticatorsMatch} predicate equals the history summary in the primed state of the Memoir-Basic NVRAM.

\[ \text{(7.5) } ll1.HistorySummary = ll1.SomeHistorySummary \]

The first step is to show the equality of the history state bindings that bind each of these history summaries to their respective state hashes.

\[ \text{(8.1) } ll1.HistoryStateBinding = ll1.SomeHistoryStateBinding \]

By hypothesis, the authenticator is a valid MAC for the garbage history state binding.

\[ \text{(9.1) } \text{ValidateMAC(} \]

\[ LL2NVRAM.symmetricKey, ll1.SomeHistoryStateBinding, ll1.Authenticator) \]

\[ \text{!}(\text{ll1.NVRAM.symmetricKey} = LL2NVRAM.symmetricKey) \]

By (2.1) \text{def } LL2Refinement

\[ \text{(10.2) } \text{QED} \]

By (7.1), (10.1)

The definition of the \text{AuthenticatorsMatch} predicate tells us that the Memoir-Basic authenticator was generated as a MAC from the history state binding.

\[ \text{(9.2) } ll1.Authenticator = GenerateMAC(LL2NVRAM.symmetricKey, ll1.HistoryStateBinding) \]

By (7.3) \text{def } ll1.HistoryStateBinding

350
The remaining preconditions are types.

(9).3. \texttt{LL2NVRAM.symmetricKey} \in \texttt{SymmetricKeyType}

\texttt{BY (2).1, LL2SubtypeImplicationLemmaDEF LL2SubtypeImplication}

(9).4. \texttt{ll1HistoryStateBinding} \in \texttt{HashType}

\texttt{BY (7).4}

(9).5. \texttt{ll1SomeHistoryStateBinding} \in \texttt{HashType}

\texttt{⟨9⟩4. ll1SomeHistorySummary} \in \texttt{HashDomain}

\texttt{⟨11⟩1. ll1SomeHistorySummary} \in \texttt{HashType}

\texttt{BY (6).1}

\texttt{⟨11⟩2. QED}

\texttt{BY ⟨11⟩1 DEF HashDomain}

(9).6. QED

\texttt{BY ⟨9⟩1, ⟨9⟩2, ⟨9⟩3, ⟨9⟩4, ⟨9⟩5, MACCollisionResistant}

By the collision resistance of the hash function, the equality of the history state bindings implies the equality of the history summaries.

(8).2. QED

(9).1. \texttt{ll1SomeHistorySummary} \in \texttt{HashDomain}

\texttt{⟨10⟩1. ll1SomeHistorySummary} \in \texttt{HashType}

\texttt{BY (6).1}

⟨10⟩2. QED

\texttt{BY ⟨10⟩1 DEF HashDomain}

(9).2. \texttt{ll1HistorySummary} \in \texttt{HashDomain}

\texttt{⟨10⟩1. ll1HistorySummary} \in \texttt{HashType}

\texttt{BY (7).3}

⟨10⟩2. QED

\texttt{BY ⟨10⟩1 DEF HashDomain}

(9).3. \texttt{stateHash1} \in \texttt{HashDomain}

\texttt{⟨10⟩1. stateHash1} \in \texttt{HashType}

\texttt{BY (6).1}

⟨10⟩2. QED

\texttt{BY ⟨10⟩1 DEF HashDomain}

(9).4. \texttt{stateHash2} \in \texttt{HashDomain}

\texttt{⟨10⟩1. stateHash2} \in \texttt{HashType}

\texttt{BY (7).3}

⟨10⟩2. QED

\texttt{BY ⟨10⟩1 DEF HashDomain}

⟨9⟩5. QED

\texttt{BY ⟨8⟩1, ⟨9⟩2, ⟨9⟩1, ⟨9⟩4, ⟨9⟩3, HashCollisionResistant}

\texttt{DEF ll1HistoryStateBinding, ll1SomeHistoryStateBinding}

We pick a value for the Memoir-Opt previous-inputs-summary existential variable inside the \texttt{HistorySummariesMatchRecursion} predicate that satisfies this predicate for (1) the Memoir-Basic’s history summary picked to satisfy the \texttt{AuthenticatorsMatch} predicate and (2) the input taken from the universal quantifier in the previous conjunct in the \texttt{LL1RestrictedCorruption} action.

(7).6. PICK \texttt{previousLL2HistorySummary} \in \texttt{HistorySummaryType} :

\texttt{HistorySummariesMatchRecursion(}

\texttt{    LL1NVRAM.historySummary},

\texttt{    LL2NVRAMLogicalHistorySummary'),

\texttt{351}
The Memoir-Basic s history summary picked to satisfy the AuthenticatorsMatch predicate matches the primed logical history summary in the Memoir-Opt NVRAM and SPCR, by the refinement and the above equality.

\[
\text{HistorySummariesMatch}(\text{LL1NVRAM.historySummary'},\text{LL2NVRAMLogicalHistorySummary'},\text{LL2NVRAM.hashBarrier'})
\]

We prove that the HistorySummariesMatch predicate equals the HistorySummariesMatchRecursion predicate in this case. We assert each condition required by the definition of the predicate.

\[
\text{HistorySummariesMatch}(\text{LL1NVRAM.historySummary'},\text{LL2NVRAMLogicalHistorySummary'},\text{LL2NVRAM.hashBarrier'}) = \text{HistorySummariesMatchRecursion}(\text{LL1NVRAM.historySummary'},\text{LL2NVRAMLogicalHistorySummary'},\text{LL2NVRAM.hashBarrier'})
\]

Finally, from HistorySummariesMatchDefinition, we can conclude that the HistorySummariesMatch predicate equals the quantified HistorySummariesMatchRecursion predicate.

\[
\text{QED}
\]
3. Pick \( \text{prevInput} \in \text{InputType}, \)
\( \text{previousLL1HistorySummary} \in \text{HashType}, \)
\( \text{previousLL2HistorySummary} \in \text{HistorySummaryType} : \)
\( \text{HistorySummariesMatchRecursion}(\)
\( \text{LL1NVRAM.historySummary'}, \)
\( \text{LL2NVRAMLLogicalHistorySummary'}, \)
\( \text{LL2NVRAM.hashBarrier'})(\)
\( \text{prevInput}, \)
\( \text{previousLL1HistorySummary}, \)
\( \text{previousLL2HistorySummary}) \)

We prove some types, to satisfy the universal quantifiers in \( \text{HistorySummariesMatchRecursion} \) predicate.

\( 1. \ \ll 1 \text{HistorySummary} \in \text{HashType} \)
\( \text{by} \ (7)3 \)
\( 2. \ \text{LL2NVRAMLLogicalHistorySummary'} \in \text{HistorySummaryType} \)
\( \text{by} \ (2)1, \ \text{LL2NVRAMLLogicalHistorySummaryTypeSafe} \)
\( 3. \ \text{LL2NVRAM.hashBarrier'} \in \text{HashType} \)
\( \text{by} \ (2)1, \ \text{LL2SubtypeImplicationLemma} \text{DEF LL2SubtypeImplication} \)
\( 4. \ \text{HistorySummariesMatchRecursion}(\)
\( \text{LL1NVRAM.historySummary'}, \)
\( \text{LL2NVRAMLLogicalHistorySummary'}, \)
\( \text{LL2NVRAM.hashBarrier'}) \)
\( \text{by} \ (8)1, \ (8)2 \)
\( 5. \ \text{QED} \)
\( \text{by} \ (9)1, \ (9)2, \ (9)4 \text{ DEF HistorySummariesMatchRecursion} \)

We prove that the existential variables for the previous input and the previous history summary in the above pick are equal to the input and history summary taken from the universal quantifiers in the previous conjunct in the \( \text{LL1RestrictedCorruption} \) action. We use the \( \text{HashCollisionResistant} \) property.

\( 1. \ \ll 1 \text{SomeHistorySummary} = \text{previousLL1HistorySummary} \)
\( \land \ \text{someInput} = \text{prevInput} \)

We prove the necessary types for the \( \text{HashCollisionResistant} \) property.

\( 1. \ \ll 1 \text{SomeHistorySummary} \in \text{HashDomain} \)
\( \text{by} \ (6)1 \)
\( 2. \ \text{QED} \)
\( \text{by} \ (10)1 \text{ DEF HashDomain} \)
\( 2. \ \text{someInput} \in \text{HashDomain} \)
\( \text{by} \ (6)1 \)
\( \text{QED} \)
\( \text{by} \ (10)1 \text{ DEF HashDomain} \)
\( 3. \ \text{previousLL1HistorySummary} \in \text{HashDomain} \)
\( \text{by} \ (8)3 \)
\( \text{QED} \)
\( \text{by} \ (10)1 \text{ DEF HashDomain} \)
\( 4. \ \text{prevInput} \in \text{HashDomain} \)
\( \text{by} \ (8)3 \)
\( \text{QED} \)
\( \text{by} \ (10)1 \text{ DEF HashDomain} \)

The hashes are equal, because each is equal to the history summary in the primed Memoir-Basic \( \text{NVRAM}. \)
The hash of the taken history summary and input are equal to the history summary in the primed Memoir-Basic NVRAM by assumption of the antecedent in the previous conjunct in the \textit{LL1RestrictedCorruption} action.

The hash of the picked history summary and input are equal to the history summary in the primed Memoir-Basic NVRAM by the definition of the \textit{HistorySummariesMatchRecursion} predicate.

One of the conjuncts in the definition of \textit{HistorySummariesMatchRecursion} is that the Memoir-Opt history summary is a successor of a previous history summary.

We re-state the definitions from the \textit{let in} the \textit{Successor} operator.

We hide the definitions.

First, we prove that the logical history summary cannot be a successor.

We re-state a definition from the \textit{let} in the \textit{Successor} operator.

We hide the definition.

There is only one sub-step, which is proving that the extension fields of these two records are unequal.

The extension field of the logical history summary is a crazy hash value, as proven above.
Second, we prove that the extension field of the successor history summary from \texttt{LL2HistorySummaryIsSuccessor} is computed from the hash function.

(9) 2. \texttt{successorHistorySummary.extension} = \\
\texttt{Hash(previousLL2HistorySummary.extension, securedInput)}

BY DEF \texttt{successorHistorySummary, Successor, securedInput}

Third, we prove that the two hashes are unequal. We will use the \texttt{CrazyHashValueUnique} property.

(9) 3. \texttt{Hash(previousLL2HistorySummary.extension, securedInput) \neq CrazyHashValue}

To employ the \texttt{CrazyHashValueUnique} property, we need to prove some types.

(10) 1. previousLL2HistorySummary.extension \in HashDomain

(11) 1. previousLL2HistorySummary.extension \in HashType

(12) 1. previousLL2HistorySummary \in HistorySummaryType

\texttt{someInput \in InputType}

\texttt{LL2NVRAM.hashBarrier' \in HashDomain}

\texttt{LL2NVRAM.hashBarrier' \in HashType}

(13) 2. QED

BY (12) 1 \texttt{def HistorySummaryType}

(11) 2. QED

BY (11) 1 \texttt{def HashDomain}

(10) 2. securedInput \in HashDomain

(11) 1. securedInput \in HashType

(12) 1. LL2NVRAM.hashBarrier' \in HashDomain

(13) 1. LL2NVRAM.hashBarrier' \in HashType

BY (2) 1, LL2SubtypeImplicationLemmaDEF LL2SubtypeImplication

(13) 2. QED

BY (13) 1 \texttt{def HashDomain}

(12) 2. someInput \in HashDomain

(13) 1. someInput \in InputType

BY (6) 1

(13) 2. QED

BY (13) 1 \texttt{def HashDomain}

(12) 3. QED

BY (12) 1, (12) 2, HashTypeSafe\texttt{def securedInput}

(11) 2. QED

BY (11) 1 \texttt{def HashDomain}

(10) 3. QED

BY (10) 1, (10) 2, CrazyHashValueUnique

(9) 4. QED

BY (9) 1, (9) 2, (9) 3

(8) 2. QED

BY (8) 1

Second, we prove that the logical history summary cannot be a checkpoint.

(7) 10. \texttt{LL2NVRAMLogicalHistorySummary' \neq checkpointedSuccessorHistorySummary}

The extension field of the logical history summary does not equal the base hash value, as we proved above.

(8) 1. \texttt{LL2NVRAMLogicalHistorySummary.extension' \neq BaseHashValue}

BY (4) 1, CrazyHashValueUnique

The extension field of the checkpointed successor does equal the base hash value. This follows from the \texttt{CheckpointHasBaseExtensionLemma}.

(8) 2. \texttt{checkpointedSuccessorHistorySummary.extension = BaseHashValue}

(9) 1. \texttt{successorHistorySummary \in HistorySummaryType}

(10) 1. previousLL2HistorySummary \in HistorySummaryType

BY (7) 6

(10) 2. \texttt{someInput \in InputType}

BY (6) 1
(10) 3. $LL2NVRAM.hashBarrier' \in HashType$

   BY (2) 1, $LL2SubtypeImplicationLemma$ def $LL2SubtypeImplication$

   (10) 4. QED

   BY (10) 1, (10) 2, (10) 3, $SuccessorTypeSafe$ def $successorHistorySummary$

   (9) 2. QED

   BY (9) 1, $CheckpointHasBaseExtensionLemma$ def $checkpointedSuccessorHistorySummary$

(8) 3. QED

   BY (8) 1, (8) 2

We thus have a contradiction.

(7) 11. QED

   BY (7) 8, (7) 9, (7) 10

(6) 4. QED

   BY (6) 3 def $llSomeHistoryStateBinding$

We prove the third conjunct within the $nvram$ conjunct of the $LL1RestrictedCorruption$ action.

(5) 4. $LL1NVRAM' = [\$
    historySummary \mapsto LL1NVRAM.historySummary',
    symmetricKey \mapsto LL1NVRAM.symmetricKey\$
]

(6) 1. $LL1NVRAM \in LL1TrustedStorageType$

   BY (2) 1 def $LL2Refinement$

(6) 2. $LL1NVRAM' \in LL1TrustedStorageType$

   BY (2) 1 def $LL2Refinement$

(6) 3. $UNCHANGED\ LL1NVRAM.symmetricKey$

   (7) 1. $UNCHANGED\ LL2NVRAM.symmetricKey$

   BY (3) 2

   (7) 2. QED

   BY (2) 1, (7) 1, $UnchangedNVRAMSymmetricKeyLemma$

(6) 4. QED

   BY (6) 1, (6) 2, (6) 3, $LL1NVRAMRecordCompositionLemma$

(5) 5. QED

   BY (5) 1, (5) 2, (5) 3, (5) 4

The $ram$ conjunct of the $LL1RestrictedCorruption$ action is satisfied because the $trashed$ disjunct is satisfied.

(4) 3. $LL1RestrictedCorruption! ram$

The $trashed$ disjunct of the $LL1RestrictedCorruption$ action is satisfied by the proof above.

(5) 1. $LL1RestrictedCorruption! ram! trashed$

   BY (3) 3

(5) 2. QED

   BY (5) 1

The disk is unchanged by the $UnchangedDiskLemma$.

(4) 4. $UNCHANGED\ LL1Disk$

   BY (2) 1, (3) 2, $UnchangedDiskLemma$

The set of available inputs is unchanged by the $UnchangedAvailableInputsLemma$.

(4) 5. $UNCHANGED\ LL1AvailableInputs$

   BY (2) 1, (3) 2, $UnchangedAvailableInputsLemma$

The set of observed outputs is unchanged by the $UnchangedObservedOutputsLemma$.

(4) 6. $UNCHANGED\ LL1ObservedOutputs$

   BY (2) 1, (3) 2, $UnchangedObservedOutputsLemma$

The set of observed authenticators is unchanged by the $UnchangedObservedAuthenticatorsLemma$.

(4) 7. $UNCHANGED\ LL1ObservedAuthenticators$

   BY (2) 1, (3) 2, $UnchangedObservedAuthenticatorsLemma$
Lastly, we tie together all of the required aspects of the $LL1_{\text{RestrictedCorruption}}$ definition.

For the ELSE case, we assume that an extension is not in progress and show that this refines to an $LL1_{\text{Restart}}$ action.

We reveal the definition of the unprimed logical history summary in the Memoir-Opt NVRAM and SPCR, given that there is no extension in progress.

We reveal the definition of the primed logical history summary in the Memoir-Opt NVRAM and SPCR, given that there is no extension in progress.

The history summary anchor in the Memoir-Opt NVRAM is unchanged by a $LL2_{\text{CorruptSPCR}}$ action.
Since the value of $LL2NVRAM_{LogicalHistorySummary}$ is determined entirely by the history summary in the Memoir-Opt $NVRAM$, and since this value is unchanged, the value of $LL2NVRAM_{LogicalHistorySummary}$ is unchanged.

(7)4. QED
   BY (7)1, (7)2, (7)3
(6)2. UNCHANGED $LL2NVRAM_{.symmetricKey}$
   BY (3)2
(6)3. UNCHANGED $LL2NVRAM_{.hashBarrier}$
   BY (3)2
(6)4. QED
   BY (2)1, (6)1, (6)2, (6)3, $UnchangedNVRAM_{HistorySummaryLemma}$
(5)2. UNCHANGED $LL1NVRAM_{.symmetricKey}$
(6)1. UNCHANGED $LL2NVRAM_{.symmetricKey}$
   BY (3)2
(6)2. QED
   BY (2)1, (6)1, $UnchangedNVRAM_{SymmetricKeyLemma}$
(5)3. $LL1NVRAM \in LL1_{TrustedStorageType}$
   BY (2)1 $Def LL2_{Refinement}$
(5)4. $LL1NVRAM' \in LL1_{TrustedStorageType}$
   BY (2)1 $Def LL2_{Refinement}$
(5)5. QED
   BY (5)1, (5)2, (5)3, (5)4, $LL1NVRAM_{RecordCompositionLemma}$

The set of available inputs is unchanged by the $UnchangedAvailableInputsLemma$.

(4)4. UNCHANGED $LL1AvailableInputs$
   BY (2)1, (3)2, $UnchangedAvailableInputsLemma$
The set of observed outputs is unchanged by the $UnchangedObservedOutputsLemma$.

(4)5. UNCHANGED $LL1ObservedOutputs$
   BY (2)1, (3)2, $UnchangedObservedOutputsLemma$
The set of observed authenticators is unchanged by the $UnchangedObservedAuthenticatorsLemma$.

(4)6. UNCHANGED $LL1ObservedAuthenticators$
   BY (2)1, (3)2, $UnchangedObservedAuthenticatorsLemma$
Lastly, we tie together all of the required aspects of the $LL1_{Restart}$ definition.

(4)7. QED
   BY (4)1, (4)2, (4)3, (4)4, (4)5, (4)6 $Def LL1_{Restart}$
Both THEN and ELSE cases are proven.

(3)6. QED
   BY (3)4, (3)5
A Memoir-Opt $LL2_{ReadDisk}$ action refines to a Memoir-Basic $LL1_{ReadDisk}$ action.

(2)8. $LL2_{ReadDisk} \Rightarrow LL1_{ReadDisk}$
(3)1. HAVE $LL2_{ReadDisk}$
The primed state of the $LL1_{RAM}$ equals the unprimed state of the $LL1_{Disk}$. The proof is somewhat tedious but completely straightforward.

(3)2. $LL1_{RAM'} = LL1_{Disk}$
   (4)1. $LL2_{RAM'} = LL2_{Disk}$
      BY (3)1 $Def LL2_{ReadDisk}$
(4)2. UNCHANGED $LL2NVRAM_{.hashBarrier}$
      BY (3)1 $Def LL2_{ReadDisk}$
The primed public state field of the RAM equals the unprimed public state field of the disk.

(4)3. $LL1_{RAM}.publicState' = LL1_{Disk}.publicState$
(5)1. $LL2_{RAM}.publicState' = LL2_{Disk}.publicState$
BY (4) 1
(5) 2. $LL1Disk.publicState = LL2Disk.publicState$
BY (2) 1 DEF $LL2Refinement$
(5) 3. $LL1RAM.publicState' = LL2RAM.publicState'$
BY (2) 1 DEF $LL2Refinement$
(5) 4. QED
BY (5) 1, (5) 2, (5) 3

The primed encrypted private state field of the RAM equals the unprimed encrypted private state field of the disk.

(4) 4. $LL1RAM.privateStateEnc' = LL1Disk.privateStateEnc$
(5) 1. $LL2RAM.privateStateEnc' = LL2Disk.privateStateEnc$
BY (4) 1
(5) 2. $LL1Disk.privateStateEnc = LL2Disk.privateStateEnc$
BY (2) 1 DEF $LL2Refinement$
(5) 3. $LL1RAM.privateStateEnc' = LL2RAM.privateStateEnc'$
BY (2) 1 DEF $LL2Refinement$
(5) 4. QED
BY (5) 1, (5) 2, (5) 3

The primed history summary field of the RAM equals the unprimed history summary field of the disk.

(4) 5. $LL1RAM.historySummary' = LL1Disk.historySummary$
(5) 1. $LL2RAM.historySummary' = LL2Disk.historySummary$
BY (4) 1
(5) 2. $HistorySummariesMatch(\$
    LL1Disk.historySummary, \$
    LL2Disk.historySummary, \$
    LL2NVRAM.hashBarrier)$
BY (2) 1 DEF $LL2Refinement$
(5) 3. $HistorySummariesMatch(\$
    LL1RAM.historySummary', \$
    LL2RAM.historySummary', \$
    LL2NVRAM.hashBarrier')$
BY (2) 1 DEF $LL2Refinement$
(5) 4. QED
BY (4) 2, (5) 1, (5) 2, (5) 3, (6) 1, (6) 2, (6) 3, (6) 4, $HistorySummariesMatchUniqueLemma$

The primed authenticator field of the RAM equals the unprimed authenticator field of the disk.

(4) 6. $LL1RAM.authenticator' = LL1Disk.authenticator$
(5) 1. $LL2RAM.authenticator' = LL2Disk.authenticator$
BY (4) 1
(5) 2. $\exists symmetricKey \in SymmetricKeyType : \$
    AuthenticatorsMatch(\$
    LL1Disk.authenticator, \$
    LL2Disk.authenticator,$
symmetricKey,
LL2NVRAM.hashBarrier)
BY (2) 1 DEF LL2Refinement
(5.3) \exists \text{symmetricKey} \in \text{SymmetricKeyType} :
\text{AuthenticatorsMatch}(
  \text{LL1RAM}.\text{authenticator}',
  \text{LL2RAM}.\text{authenticator}',
  \text{symmetricKey},
  \text{LL2NVRAM}.\text{hashBarrier}')
BY (2) 1 DEF LL2Refinement
(5.4) QED
(6.1) \text{LL1Disk}.\text{authenticator} \in \text{MACType}
BY (2) 1 DEF LL2Refinement, LL1UntrustedStorageType
(6.2) \text{LL1RAM}.\text{authenticator}' \in \text{MACType}
BY (2) 1 DEF LL2Refinement, LL1UntrustedStorageType
(6.3) \text{LL2Disk}.\text{authenticator} \in \text{MACType}
BY (2) 1, LL2SubtypeImplicationLemma\text{DEF LL2SubtypeImplication}
(6.4) \text{LL2NVRAM}.\text{hashBarrier} \in \text{HashType}
BY (2) 1, LL2SubtypeImplicationLemma\text{DEF LL2SubtypeImplication}
(6.5) QED
BY (4) 2, (5) 1, (5) 2, (5) 3, (6) 1, (6) 2, (6) 3, (6) 4,
\text{AuthenticatorsMatchUniqueLemma}, \text{LL1DiskRecordCompositionLemma}
\text{LL1RAM}' \in \text{LL1UntrustedStorageType}
BY (2) 1 DEF LL2Refinement
(4.8) \text{LL1Disk} \in \text{LL1UntrustedStorageType}
BY (2) 1 DEF LL2Refinement
(4.9) QED
BY (4) 3, (4) 4, (4) 5, (4) 6, (4) 7, (4) 8,
\text{LL1RAM}' \in \text{LL1UntrustedStorageType}, \text{LL1DiskRecordCompositionLemma}
\text{LL1Disk}' \in \text{LL1UntrustedStorageType}
BY (2) 1 DEF LL2Refinement
\text{LL1Disk} \in \text{LL1UntrustedStorageType}
BY (2) 1 DEF LL2Refinement
\text{LL1Disk}' \in \text{LL1UntrustedStorageType}
BY (2) 1 DEF LL2Refinement
\text{LL1Disk}' \in \text{LL1UntrustedStorageType}
BY (2) 1, LL1UntrustedStorageTypeLemma\text{DEF LL2ReadDisk}
(3.5) \text{UNCHANGED LL1Disk}
BY (2) 1, (3) 1, UnchangedDiskLemma\text{DEF LL2ReadDisk}
(3.4) \text{UNCHANGED LL1NVRAM}
BY (2) 1, (3) 1, UnchangedNVRAMLemma\text{DEF LL2ReadDisk}
(3.6) \text{UNCHANGED LL1AvailableInputs}
BY (2) 1, (3) 1, UnchangedAvailableInputsLemma\text{DEF LL2ReadDisk}
(3.7) \text{UNCHANGED LL1ObservedOutputs}
BY (2) 1, (3) 1, UnchangedObservedOutputsLemma\text{DEF LL2ReadDisk}
(3.8) QED
BY (3) 2, (3) 3, (3) 4, (3) 5, (3) 6, (3) 7 DEF LL1ReadDisk
A Memoir-Opt LL2WriteDisk action refines to a Memoir-Basic LL1WriteDisk action.
(2) 9. LL2WriteDisk \Rightarrow LL1WriteDisk
(3) 1. HAVE LL2WriteDisk
The primed state of the LL1Disk equals the unprimed state of the LL1RAM. The proof is somewhat tedious but completely straightforward.
(3) 2. LL1Disk' = LL1RAM
(4) 1. LL2Disk' = LL2RAM
BY (3) 1 DEF LL2WriteDisk
(4) 2. \land \text{UNCHANGED LL2NVRAM.symmetricKey}
\land \text{UNCHANGED LL2NVRAM.hashBarrier}
The primed authenticator field of the disk equals the unprimed authenticator field of the RAM.

\((4.3)\)  \(\text{LL1Disk.authenticator}' = \text{LL1RAM.authenticator} \)

\((5.1)\)  \(\text{LL2Disk.authenticator}' = \text{LL2RAM.authenticator} \)

\(\text{by \ (4.1)}\)

\((5.2)\)  \(\text{LL1RAM.authenticator} = \text{LL2RAM.authenticator} \)

\(\text{by \ (2.1) \text{\textit{LL2Refinement}}}\)

\((5.3)\)  \(\text{LL1Disk.authenticator}' = \text{LL2Disk.authenticator}' \)

\(\text{by \ (2.1) \text{\textit{LL2Refinement}}}\)

\((5.4)\)  \text{QED}\n
\(\text{by \ (5.1), (5.2), (5.3)}\)

The primed encrypted private state field of the disk equals the unprimed encrypted private state field of the RAM.

\((4.4)\)  \(\text{LL1Disk.privateStateEnc}' = \text{LL1RAM.privateStateEnc} \)

\((5.1)\)  \(\text{LL2Disk.privateStateEnc}' = \text{LL2RAM.privateStateEnc} \)

\(\text{by \ (4.1)}\)

\((5.2)\)  \(\text{LL1RAM.privateStateEnc} = \text{LL2RAM.privateStateEnc} \)

\(\text{by \ (2.1) \text{\textit{LL2Refinement}}}\)

\((5.3)\)  \(\text{LL1Disk.privateStateEnc}' = \text{LL2Disk.privateStateEnc}' \)

\(\text{by \ (2.1) \text{\textit{LL2Refinement}}}\)

\((5.4)\)  \text{QED}\n
\(\text{by \ (5.1), (5.2), (5.3)}\)

The primed history summary field of the disk equals the unprimed history summary field of the RAM.

\((4.5)\)  \(\text{LL1Disk.historySummary}' = \text{LL1RAM.historySummary} \)

\((5.1)\)  \(\text{LL2Disk.historySummary}' = \text{LL2RAM.historySummary} \)

\(\text{by \ (4.1)}\)

\((5.2)\)  \text{\textit{HistorySummariesMatch}}(\text{LL1RAM.historySummary, LL2RAM.historySummary, LL2NVRAM.hashBarrier})

\(\text{by \ (2.1) \text{\textit{LL2Refinement}}}\)

\((5.3)\)  \text{\textit{HistorySummariesMatch}}(\text{LL1Disk.historySummary}', \text{LL2Disk.historySummary}', \text{LL2NVRAM.hashBarrier}')

\(\text{by \ (2.1) \text{\textit{LL2Refinement}}}\)

\((5.4)\)  \text{QED}\n
\((6.1)\)  \text{LL1RAM.historySummary} \in \text{HashType}

\(\text{by \ (2.1) \text{\textit{LL2Refinement}}, \text{LL1UntrustedStorageType}}\)

\((6.2)\)  \text{LL1Disk.historySummary}' \in \text{HashType}

\(\text{by \ (2.1) \text{\textit{LL2Refinement}}, \text{LL1UntrustedStorageType}}\)

\((6.3)\)  \text{LL2RAM.historySummary} \in \text{HistorySummaryType}

\(\text{by \ (2.1), \text{LL2SubtypeImplicationLemma} \text{\textit{LL2SubtypeImplication}}}\)

\((6.4)\)  \text{LL2NVRAM.hashBarrier} \in \text{HashType}

\(\text{by \ (2.1), \text{LL2SubtypeImplicationLemma} \text{\textit{LL2SubtypeImplication}}}\)

\((6.5)\)  \text{QED}\n
\(\text{by \ (4.2), (5.1), (5.2), (5.3), (6.1), (6.2), (6.3), (6.4), \text{HistorySummariesMatchUniqueLemma}}\)

The primed authenticator field of the disk equals the unprimed authenticator field of the RAM.

\((4.6)\)  \(\text{LL1Disk.authenticator}' = \text{LL1RAM.authenticator} \)

\((5.1)\)  \(\text{LL2Disk.authenticator}' = \text{LL2RAM.authenticator} \)

\(\text{by \ (4.1)}\)
(5)2. \( \exists \text{symmetricKey} \in \text{SymmetricKeyType} : \)
    \[
    \begin{align*}
    \text{AuthenticatorsMatch}( & \\
    \text{LL1RAM.authenticator}, & \\
    \text{LL2RAM.authenticator}, & \\
    \text{symmetricKey}, & \\
    \text{LL2NVRAM.hashBarrier}) & \\
    \end{align*}
    \]
    \text{BY \( \langle 2 \rangle 1 \) DEF \text{LL2Refinement}}

(5)3. \( \exists \text{symmetricKey} \in \text{SymmetricKeyType} : \)
    \[
    \begin{align*}
    \text{AuthenticatorsMatch}( & \\
    \text{LL1Disk.authenticator}', & \\
    \text{LL2Disk.authenticator}', & \\
    \text{symmetricKey}, & \\
    \text{LL2NVRAM.hashBarrier'}) & \\
    \end{align*}
    \]
    \text{BY \( \langle 2 \rangle 1 \) DEF \text{LL2Refinement}}

(5)4. QED
    \[
    \begin{align*}
    \text{BY \( \langle 4 \rangle 2, \langle 5 \rangle 1, \langle 5 \rangle 2, \langle 5 \rangle 3, \langle 6 \rangle 1, \langle 6 \rangle 2, \langle 6 \rangle 3, \langle 6 \rangle 4, \)} & \\
    \text{AuthenticatorsMatchUniqueLemma} & \\
    \text{BY \( \langle 2 \rangle 1 \) DEF \text{LL2Refinement}} & \\
    \text{BY \( \langle 2 \rangle 1 \) DEF \text{LL2Refinement}} & \\
    \text{BY \( \langle 2 \rangle 1 \) DEF \text{LL2Refinement}} & \\
    \text{BY \( \langle 4 \rangle 3, \langle 4 \rangle 4, \langle 4 \rangle 5, \langle 4 \rangle 6, \langle 4 \rangle 7, \langle 4 \rangle 8, \)} & \\
    \text{LL1DiskRecordCompositionLemma, LL1RAMRecordCompositionLemma} & \\
    \end{align*}
    \]

All variables other than \text{LL1Disk} are unchanged, by virtue of the corresponding lemmas.

(3)3. \text{UNCHANGED LL1RAM}
    \text{BY \( \langle 2 \rangle 1, \langle 3 \rangle 1, \text{UNCHANGEDRAMLemma} \) DEF \text{LL2WriteDisk}}

(3)4. \text{UNCHANGED LL1NVRAM}
    \text{BY \( \langle 2 \rangle 1, \langle 3 \rangle 1, \text{UNCHANGEDNVRAMLemma} \) DEF \text{LL2WriteDisk}}

(3)5. \text{UNCHANGED LL1AvailableInputs}
    \text{BY \( \langle 2 \rangle 1, \langle 3 \rangle 1, \text{UNCHANGEDAvailableInputsLemma} \) DEF \text{LL2WriteDisk}}

(3)6. \text{UNCHANGED LL1ObservedOutputs}
    \text{BY \( \langle 2 \rangle 1, \langle 3 \rangle 1, \text{UNCHANGEDObservedOutputsLemma} \) DEF \text{LL2WriteDisk}}

(3)7. \text{UNCHANGED LL1ObservedAuthenticators}
    \text{BY \( \langle 2 \rangle 1, \langle 3 \rangle 1, \text{UNCHANGEDObservedAuthenticatorsLemma} \) DEF \text{LL2WriteDisk}}

(3)8. QED
    \text{BY \( \langle 3 \rangle 2, \langle 3 \rangle 3, \langle 3 \rangle 4, \langle 3 \rangle 5, \langle 3 \rangle 6, \langle 3 \rangle 7 \) DEF \text{LL1WriteDisk}}

\text{A Memoir-Opt \text{LL2CorruptRAM} action refines to a Memoir-Basic \text{LL1CorruptRAM} action.}

(2)10. \text{LL2CorruptRAM } \Rightarrow \text{LL1CorruptRAM}
    \text{We assume the antecedent.}

(3)1. \text{HAVE LL2CorruptRAM}
    \text{We pick a set of variables of the appropriate types that satisfy the \text{LL2CorruptRAM} action.}
We then prove the disjunction regarding the authenticator field in the primed LL1RAM record.

We pick a symmetric key that satisfies the AuthenticatorsMatch predicate for the primed states of the authenticators in the RAM variables of the two specs.

We pick a set of variables of the appropriate types that satisfy the quantified AuthenticatorsMatch predicate.

We re-state the definitions from the let in AuthenticatorsMatch.

We hide the definitions.

We prove each aspect of LL1CorruptRAM separately. First, we prove the types of the three existentially quantified variables in the definition of LL1CorruptRAM.

We then prove the disjunction regarding the authenticator field in the primed LL1RAM record.
(3)9. $\forall \text{LL1RAM.authenticator}' \in
\text{LL1ObservedAuthenticators}$
$\forall \text{LL1RAM.authenticator}' = \text{GenerateMAC(ll2FakeSymmetricKey, ll1HistoryStateBinding)}$

We prove that when the authenticator in the Memoir-Opt spec is in the set of observed authenticators, then the authenticator in the Memoir-Basic spec is in the set of observed authenticators. This follows directly from the AuthenticatorInSetLemma, once we prove the preconditions for the lemma.

(4)1. Assume $\text{ll2UntrustedStorage.authenticator} \in \text{LL2ObservedAuthenticators}$
Prove $\text{LL1RAM.authenticator}' \in \text{LL1ObservedAuthenticators}$

We need to prove some types.

(5)1. $\text{LL1RAM.authenticator}' \in \text{MACType}$
(6)1. $\text{LL1RAM'} \in \text{LL1UntrustedStorageType}$
By (2)1 DEF LL2Refinement
(6)2. Qed
By (6)1 DEF LL1UntrustedStorageType
(5)2. $\text{LL2RAM.authenticator}' \in \text{MACType}$
By (2)1, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication
(5)3. $\text{LL1ObservedAuthenticators} \in \text{SUBSET MACType}$
By (2)1 DEF LL2Refinement
(5)4. $\text{LL2ObservedAuthenticators} \in \text{SUBSET MACType}$
By (2)1 DEF LL2TypeInvariant
(5)5. $\text{LL2NVRAM.symmetricKey} \in \text{SymmetricKeyType}$
By (2)1, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication
(5)6. $\text{LL2NVRAM.hashBarrier} \in \text{HashType}$
By (2)1, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication

Then we prove the three conjuncts in the antecedent of the AuthenticatorInSetLemma. The first conjunct follows from the refinement.

(5)7. $\exists \text{symmetricKey1} \in \text{SymmetricKeyType}$:
AuthenticatorsMatch (\begin{align*}
\text{LL1RAM.authenticator}', \\
\text{LL2RAM.authenticator}', \\
\text{symmetricKey1}, \\
\text{LL2NVRAM.hashBarrier})
\end{align*})
(6)1. $\exists \text{symmetricKey1} \in \text{SymmetricKeyType}$:
AuthenticatorsMatch (\begin{align*}
\text{LL1RAM.authenticator}', \\
\text{LL2RAM.authenticator}', \\
\text{symmetricKey1}, \\
\text{LL2NVRAM.hashBarrier}')
\end{align*})
By (2)1 DEF LL2Refinement
(6)2. UNCHANGED $\text{LL2NVRAM.hashBarrier}$
By (3)2
(6)3. Qed
By (6)1, (6)2

The second conjunct in the antecedent of the AuthenticatorInSetLemma also follows from the refinement.

(5)8. AuthenticatorSetsMatch (\begin{align*}
\text{LL1ObservedAuthenticators}, \\
\text{LL2ObservedAuthenticators}, \\
\text{LL2NVRAM.symmetricKey}, \\
\text{LL2NVRAM.hashBarrier})
\end{align*})
BY (2)1 DEF LL2Refinement

The third conjunct in the antecedent of the AuthenticatorInSetLemma follows from the first disjunct in the LL2CorruptRAM action.

(5)9. LL2RAM. authenticator’ ∈ LL2ObservedAuthenticators
   (6)1. LL2RAM. authenticator’ = ll2UntrustedStorage. authenticator
   (7)1. LL2RAM’ = ll2UntrustedStorage
        BY (3)2
   (7)2. QED
        BY (7)1
(6)2. ll2UntrustedStorage. authenticator ∈ LL2ObservedAuthenticators
        BY (4)1
   (6)3. QED
        BY (6)1, (6)2

We then invoke the AuthenticatorInSetLemma directly.

(5)10. QED
       BY (5)1, (5)2, (5)3, (5)4, (5)5, (5)6, (5)7, (5)8, (5)9, AuthenticatorInSetLemma

We prove that when the authenticator in the Memoir-Opt spec is generated as a MAC with a fake symmetric key, then the authenticator in the Memoir-Basic spec is generated as a MAC with a fake symmetric key. This follows directly from the AuthenticatorGeneratedLemma, once we prove the preconditions for the lemma.

(4)2. ASSUME ll2UntrustedStorage. authenticator = GenerateMAC(ll2FakeSymmetricKey, ll2Hash)
       PROVE LL1RAM. authenticator’ =
              GenerateMAC(ll2FakeSymmetricKey, ll1HistoryStateBinding)

We need to prove some types.

(5)1. stateHash ∈ HashType
       BY (3)4
(5)2. ll1HistorySummary ∈ HashType
       BY (3)4
(5)3. ll2HistorySummary ∈ HistorySummaryType
       BY (3)4
(5)4. LL1RAM. authenticator’ ∈ MACType
       (6)1. LL1RAM’ ∈ ll1UntrustedStorageType
             BY (2)1 DEF LL2Refinement
       (6)2. QED
             BY (6)1 DEF ll1UntrustedStorageType
(5)5. LL2RAM. authenticator’ ∈ MACType
       BY (2)1, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication
(5)6. ll2FakeSymmetricKey ∈ SymmetricKeyType
       (6)1. ll2FakeSymmetricKey ∈ SymmetricKeyType \ {LL2NVRAM. symmetricKey}
             BY (3)2
       (6)2. QED
             BY (6)1
(5)7. LL2NVRAM. hashBarrier’ ∈ HashType
             BY (2)1, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication

Then we prove the three conjuncts in the antecedent of the AuthenticatorGeneratedLemma. The first conjunct follows from a conjunct in the refinement.

(5)8. HistorySummariesMatch(ll1HistorySummary, ll2HistorySummary, LL2NVRAM. hashBarrier’)
       BY (3)4

The second conjunct in the antecedent of the AuthenticatorGeneratedLemma mainly follows from the refinement, but we also have to prove that the symmetric key specified by an existential in the refinement matches the fake symmetric key specified in the LL2CorruptRAM action. This follows from the MACUnforgeable property.
(5.9) \text{AuthenticatorsMatch}(
\begin{align*}
& \text{LL1RAM}.\text{authenticator}^\prime, \\
& \text{LL2RAM}.\text{authenticator}^\prime, \\
& ll2\text{FakeSymmetricKey}, \\
& \text{LL2NVRAM}.\text{hashBarrier}^\prime
\end{align*}
)

The main precondition for the \text{MACUnforgeable} property is that the generated MAC is validated. We first prove the validation, which follows from the \text{LL2CorruptRAM} action.

(6.1) \text{ValidateMAC}(\text{symmetricKey}, ll2\text{HistoryStateBinding}, \text{LL2RAM}.\text{authenticator}^\prime)

\begin{align*}
\text{by (3)4: DEF } ll2\text{HistoryStateBinding}, ll2\text{HistorySummaryHash}
\end{align*}

We then prove the generation, which follows from the \text{LL2CorruptRAM} action.

(6.2) \text{LL2RAM}.\text{authenticator}^\prime = \text{GenerateMAC}(ll2\text{FakeSymmetricKey}, ll2\text{Hash})

(7.1) \text{LL2RAM}^\prime = ll2\text{UntrustedStorage}

\begin{align*}
\text{by (3)2}
\end{align*}

(7.2) ll2\text{UntrustedStorage}.\text{authenticator} = \text{GenerateMAC}(ll2\text{FakeSymmetricKey}, ll2\text{Hash})

\begin{align*}
\text{by (3)2}
\end{align*}

(7.3) QED

\begin{align*}
\text{by (7.1), (7.2)}
\end{align*}

The remaining preconditions are types.

(6.3) \text{symmetricKey} \in \text{SymmetricKeyType}

\begin{align*}
\text{by (3)3}
\end{align*}

(6.4) ll2\text{FakeSymmetricKey} \in \text{SymmetricKeyType}

(7.1) ll2\text{FakeSymmetricKey} \in \text{SymmetricKeyType} \setminus \{\text{LL2NVRAM}.\text{symmetricKey}\}

\begin{align*}
\text{by (3)2}
\end{align*}

(7.2) QED

\begin{align*}
\text{by (7.1)}
\end{align*}

(6.5) ll2\text{HistoryStateBinding} \in \text{HashType}

\begin{align*}
\text{by (3)5}
\end{align*}

(6.6) ll2\text{Hash} \in \text{HashType}

\begin{align*}
\text{by (3)2}
\end{align*}

The \text{MACUnforgeable} property tells us that the two keys are equal.

(6.7) \text{symmetricKey} = ll2\text{FakeSymmetricKey}

\begin{align*}
\text{by (6.1), (6.2), (6.3), (6.4), (6.5), (6.6), MACUnforgeable}
\end{align*}

(6.8) QED

\begin{align*}
\text{by (3)3, (6)7}
\end{align*}

The third conjunct in the antecedent of the \text{AuthenticatorGeneratedLemma} mainly follows from the \text{LL2CorruptRAM} action, but we also have to prove that the history state binding specified in the refinement matches the arbitrary hash specified in the \text{LL2CorruptRAM} action.

(5.10) \text{LL2RAM}.\text{authenticator}^\prime = \text{GenerateMAC}(ll2\text{FakeSymmetricKey}, ll2\text{HistoryStateBinding})

The state authenticaton in the primed Memoir-Opt RAM equals the authenticator specified by the existential \text{ll2UntrustedStorage} in the \text{LL2CorruptRAM} action.

(6.1) \text{LL2RAM}.\text{authenticator}^\prime = ll2\text{UntrustedStorage}.\text{authenticator}

(7.1) \text{LL2RAM}^\prime = ll2\text{UntrustedStorage}

\begin{align*}
\text{by (3)2}
\end{align*}

(7.2) QED

\begin{align*}
\text{by (7.1)}
\end{align*}

The authenticator specified by the existential \text{ll2UntrustedStorage} in the \text{LL2CorruptRAM} action is generated as a MAC of the history state binding from the \text{AuthenticatorsMatch} predicate invoked by the refinement. This follows from the \text{MACCollisionResistant} property.

(6.2) ll2\text{UntrustedStorage}.\text{authenticator} = \text{GenerateMAC}(ll2\text{FakeSymmetricKey}, ll2\text{HistoryStateBinding})
The main precondition for the MACUnforgeable property is that the generated MAC is validated. We first prove the validation, which follows from the refinement.

\begin{enumerate}
\item ValidateMAC\(\text{symmetricKey}, \text{ll2HistoryStateBinding}, \text{ll2RAM.authenticator}'\)
\end{enumerate}

\text{by (3)2}

\begin{enumerate}
\item \text{ll2UntrustedStorage.authenticator} = GenerateMAC(\text{ll2FakeSymmetricKey}, \text{ll2Hash})
\end{enumerate}

\text{by (3)2}

\begin{enumerate}
\item QED
\end{enumerate}

\text{by (8)1, (8)2}

The remaining preconditions are types.

\begin{enumerate}
\item \text{symmetricKey} \in \text{SymmetricKeyType}
\end{enumerate}

\text{by (3)3}

\begin{enumerate}
\item \text{ll2FakeSymmetricKey} \in \text{SymmetricKeyType}
\end{enumerate}

\text{by (3)2}

\begin{enumerate}
\item QED
\end{enumerate}

\text{by (8)1}

\begin{enumerate}
\item \text{ll2HistoryStateBinding} \in \text{HashType}
\end{enumerate}

\text{by (3)5}

\begin{enumerate}
\item \text{ll2Hash} \in \text{HashType}
\end{enumerate}

\text{by (3)2}

The MACCollisionResistant property tells us that the two hash values are equal.

\begin{enumerate}
\item \text{ll2Hash} = \text{ll2HistoryStateBinding}
\end{enumerate}

\text{by (7)1, (7)2, (7)3, (7)4, (7)5, (7)6, MACCollisionResistant}

\begin{enumerate}
\item QED
\end{enumerate}

\text{by (4)2, (7)7}

\begin{enumerate}
\item QED
\end{enumerate}

\text{by (6)1, (6)2}

We then invoke the AuthenticatorGeneratedLemma directly.

\begin{enumerate}
\item QED
\end{enumerate}

\text{by (5)1, (5)2, (5)3, (5)4, (5)5, (5)6, (5)7, (5)8, (5)9, (5)10, AuthenticatorGeneratedLemma}

\text{DEF ll1HistoryStateBinding, ll2HistorySummaryHash, ll2HistoryStateBinding}

From the definition of LL2CorruptRAM, the above two cases are exhaustive.

\begin{enumerate}
\item V \text{ll2UntrustedStorage.authenticator} \in \text{ll2ObservedAuthenticators}
\item V \text{ll2UntrustedStorage.authenticator} = GenerateMAC(\text{ll2FakeSymmetricKey}, \text{ll2Hash})
\end{enumerate}

\text{by (3)2}

\begin{enumerate}
\item QED
\end{enumerate}

\text{by (4)1, (4)2, (4)3}

All variables other than LL1RAM are unchanged, by virtue of the corresponding lemmas.

\begin{enumerate}
\item UNCHANGED LL1Disk
\end{enumerate}

\text{by (2)1, (3)2, UnchangedDiskLemmaDEF LL2CorruptRAM}

\begin{enumerate}
\item UNCHANGED LL1NVRAM
\end{enumerate}

\text{by (2)1, (3)2, UnchangedNVRAMLemmaDEF LL2CorruptRAM}

\begin{enumerate}
\item UNCHANGED LL1AvailableInputs
\end{enumerate}

\text{by (3)11, (3)12, UnchangedAvailableInputsLemmaDEF LL2CorruptRAM}
By (2)1, (3)2, \( UnchangedAvailableInputsLemma \) we define \( LL2CorruptRAM \)

(3)13. \textbf{UNCHANGED LL1ObservedOutputs} 

By (2)1, (3)2, \( UnchangedObservedOutputsLemma \) we define \( LL2CorruptRAM \)

(3)14. \textbf{UNCHANGED LL1ObservedAuthenticators} 

By (2)1, (3)2, \( UnchangedObservedAuthenticatorsLemma \) we define \( LL2CorruptRAM \)

Lastly, we tie together all of the required aspects of the \( LL1CorruptRAM \) definition.

(3)15. \textbf{QED} 

By (3)6, (3)7, (3)8, (3)9, (3)10, (3)11, (3)12, (3)13, (3)14 we define \( LL1CorruptRAM \)

A Memoir-Opt \( LL2CorruptSPCR \) action refines to one of two actions in the Memoir-Basic spec. If an extension is in progress at the time the \( LL2Restart \) occurs, the corruption of the \( SPCR \) value caused by the \( LL2CorruptSPCR \) is fatal, so this refines to a \( LL1RestrictedCorruption \) action in the Memoir-Basic spec, which in turn refines to an \( HLDie \) action in the high-level spec. On the other hand, if an extension is not in progress at the time the \( LL2Restart \) occurs, the action refines to a stuttering step in the Memoir-Basic spec.

(2)11. \( LL2CorruptSPCR \Rightarrow \)

\[
\text{if } LL2NVRAM.extensionInProgress \text{ then }
\]

\[
LL1RestrictedCorruption
\]

\[
\text{else }
\]

\[
\text{UNCHANGED LL1Vars}
\]

We assume the antecedent.

(3)1. \textbf{HAVE LL2CorruptSPCR} 

We pick a fake hash that satisfies the \( LL2CorruptSPCR \) action.

(3)2. \textbf{PICK fakeHash ∈ HashDomain : LL2CorruptSPCR! (fakeHash)} 

By (3)1 we define \( LL2CorruptSPCR \)

We re-state the definition from \( LL2CorruptSPCR \).

(3) newHistorySummaryExtension \( \triangleq Hash(LL2SPCR, fakeHash) \)

We then hide the definition.

(3) \textbf{HIDE DEF newHistorySummaryExtension} 

Since the refinement is an if-then-else, we prove the then and else cases separately. For the then case, we assume that an extension is in progress and show that this refines to a \( LL1RestrictedCorruption \) action.

(3)3. \textbf{ASSUME LL2NVRAM.extensionInProgress} 

\textbf{PROVE LL1RestrictedCorruption}

One fact that will be useful in several places below is that the extension field in the logical history summary of the Memoir-Opt \( NVRAM \) and \( SPCR \) is not equal to the base hash value. We'll separately prove the two cases of whether or not the primed \( SPCR \) equals the base hash value.

(4)1. \( LL2NVRAM\_LogicalHistorySummary.extension' ≠ BaseHashValue \)

One fact that will be useful for both cases is that an extension is in progress in the primed state.

(5)1. \( LL2NVRAM.extensionInProgress' \)

\[
\text{(6)1. LL2NVRAM.extensionInProgress}
\]

By (3)3

\[
\text{(6)2. UNCHANGED LL2NVRAM.extensionInProgress}
\]

By (3)2

\[
\text{(6)3. QED}
\]

By (6)1, (6)2

The first case is fairly simple. When an extension is in progress but the \( SPCR \) equals the base hash value, the logical history summary equals a crazy hash value, because this situation should never arise during normal operation.

(5)2. \textbf{CASE LL2SPCR' = BaseHashValue} 

\[
\text{(6)1. LL2NVRAMLogicalHistorySummary.extension' = CrazyHashValue}
\]

(7)1. \( LL2NVRAM\_LogicalHistorySummary' = | \)
anchor \mapsto LL2NVRAM.historySummaryAnchor', \\
extension \mapsto CrazyHashValue

by (5)1, (5)2 DEF LL2NVRAMLogicalHistorySummary

(7)2. QED

by (7)1

(6)2. CrazyHashValue \neq BaseHashValue

by CrazyHashValueUnique

(6)3. QED

by (6)1, (6)2

The second case is slightly more involved. We will prove this in two steps.

(5)3. CASE LL2SPCR' \neq BaseHashValue

First, we prove that the primed value of the extension field in the Memoir-Opt logical history summary equals the new history summary extension defined in the LL2CorruptSPCR action. This follows fairly directly from the definitions of the LL2CorruptSPCR action and the LL2NVRAMLogicalHistorySummary operator.

(6)1. LL2NVRAMLogicalHistorySummary'.extension' = newHistorySummaryExtension

(7)1. LL2SPCR' = newHistorySummaryExtension

by (3)2 DEF newHistorySummaryExtension

(7)2. LL2NVRAMLogicalHistorySummary' = [
anchor \mapsto LL2NVRAM.historySummaryAnchor',
extension \mapsto LL2SPCR']

by (5)1, (5)3 DEF LL2NVRAMLogicalHistorySummary

(7)3. QED

by (7)1, (7)2

Second, we prove that the new history summary extension is not equal to the base hash value. This follows because the new history summary extension is generated as a hash by the LL2CorruptSPCR action, and the BaseHashValueUnique property tells us that no hash value generated by the Hash function can equal the base hash value.

(6)2. newHistorySummaryExtension' \neq BaseHashValue

(7)1. LL2SPCR' \in HashDomain

by (8)1 DEF LL2TypeInvariant

(8)2. QED

by (8)1 DEF HashDomain

(7)2. fakeHash \in HashDomain

by (3)2

(7)3. QED

by (7)1, (7)2, BaseHashValueUnique DEF newHistorySummaryExtension

(6)3. QED

by (6)1, (6)2

The two cases are exhaustive.

(5)4. QED

by (5)2, (5)3

We prove each conjunct of LL1RestrictedCorruption separately. First, we prove the conjunct relating to the NVRAM.

(4)2. LL1RestrictedCorruption!nvram

The primed value of the history summary field in the Memoir-Basic NVRAM serves as our witness for the garbage history summary.

(5)1. LL1NVRAM.historySummary' \in HashType

by (2)1 DEF LL2Refinement, LL1TrustedStorageType

We prove that the constraint labeled current in the LL1RestrictedCorruption action is satisfied.

(5)2. LL1RestrictedCorruption!nvram!current(LL1NVRAM.historySummary')

369
To prove the universally quantified expression, we take a set of variables of the appropriate types.

(6)1. TAKE stateHash1 ∈ HashType,
   ll1Authenticator ∈ LL1ObservedAuthenticators

We re-state the definition from within the LL1RestrictedCorruption!nram!current clause.

(6) ll1GarbageHistoryStateBinding ≜ Hash(LL1NVRAM.historySummary', stateHash1)

We hide the definition.

(6) HIDE DEF ll1GarbageHistoryStateBinding

We need to prove the nram::current conjunct, which asserts that the authenticator is not a valid MAC for the history state binding formed from the history summary in the NVRAM and any state hash.

(6)2. ¬ValidateMAC(
   LL1NVRAM.symmetricKey,
   ll1GarbageHistoryStateBinding,
   ll1Authenticator)

We will use proof by contradiction.

(7)1. SUFFICES
   ASSUME
   ValidateMAC(
   LL1NVRAM.symmetricKey,
   ll1GarbageHistoryStateBinding,
   ll1Authenticator)
   PROVE
   FALSE
   OBVIOUS

We first pick, from the set of Memoir-Opt observed authenticators, a Memoir-Opt authenticator that matches the Memoir-Basic authenticator. We know that such a authenticator exists, because the refinement asserts that the sets of observed authenticators match across the two specs.

(7)2. PICK ll2Authenticator ∈ LL2ObservedAuthenticators :
   AuthenticatorsMatch(
   ll1Authenticator,
   ll2Authenticator,
   LL2NVRAM.symmetricKey,
   LL2NVRAM.hashBarrier)

(8)1. ll1Authenticator ∈ LL1ObservedAuthenticators
   BY (6)1

(8)2. AuthenticatorSetsMatch(
   LL1ObservedAuthenticators,
   LL2ObservedAuthenticators,
   LL2NVRAM.symmetricKey,
   LL2NVRAM.hashBarrier)
   BY (2)1 DEF LL2Refinement

(8)3. QED
   BY (8)1, (8)2 DEF AuthenticatorSetsMatch

We pick a set of variables of the appropriate types that satisfy the quantified AuthenticatorsMatch predicate.

(7)3. PICK stateHash2 ∈ HashType,
   ll1HistorySummary ∈ HashType,
   ll2HistorySummary ∈ HistorySummaryType :
   AuthenticatorsMatch(
   ll1Authenticator,
   ll2Authenticator,
   LL2NVRAM.symmetricKey,
   LL2NVRAM.hashBarrier)
We prove that the Memoir-Basic's history summary picked to satisfy the AuthenticatorsMatch predicate equals the history summary in the primed state of the Memoir-Basic NVRAM.

The first step is to show the equality of the history state bindings that bind each of these history summaries to their respective state hashes.

By hypothesis, the authenticator is a valid MAC for the garbage history state binding.

The definition of the AuthenticatorsMatch predicate tells us that the Memoir-Basic authenticator was generated as a MAC from the history state binding.

The remaining preconditions are types.

By the collision resistance of the hash function, the equality of the history state bindings implies the equality of the history summaries.
The Memoir-Basic s history summary picked to satisfy the AuthenticatorsMatch predicate matches the primed logical history summary in the Memoir-Opt NVRAM and SPCR, by the refinement and the above equality.

We prove that the HistorySummariesMatch predicate equals the HistorySummariesMatchRecursion predicate in this case. We assert each condition required by the definition of the predicate.

We prove some types, to satisfy the universal quantifiers in HistorySummariesMatchDefinition.

We prove that the Memoir-Opt logical history summary does not equal the initial history summary.
(9.1) \( \text{LL2NVRAML} \text{Log} \text{icalHistorySummary} . \text{extension'} \neq \text{BaseHashValue} \)
    BY (4) 1
(9.2) \( \text{ll2InitialHistorySummary} . \text{extension} = \text{BaseHashValue} \)
    BY DEF ll2InitialHistorySummary
(9.3) QED
    BY (9) 1, (9) 2

Finally, from HistorySummariesMatchDefinition, we can conclude that the HistorySummariesMatch predicate equals the quantified HistorySummariesMatchRecursion predicate.

(8) 5. QED
    BY (8) 1, (8) 2, (8) 3, (8) 4, HistorySummariesMatchDefinition
    DEF ll2InitialHistorySummary

We pick values for the existential variables inside the HistorySummariesMatchRecursion predicate that satisfy the predicate. We know such variables exist, because the predicate is satisfied by the two previous steps.

(7) 8. PICK prevInput ∈ InputType,
    previousLL1HistorySummary ∈ HashType,
    previousLL2HistorySummary ∈ HistorySummaryType :
    HistorySummariesMatchRecursion(
        ll1HistorySummary,
        LL2NVRAMLLogicalHistorySummary',
        LL2NVRAM . hashBarrier' )!
        (prevInput,
        previousLL1HistorySummary,
        previousLL2HistorySummary)

We prove some types, to satisfy the universal quantifiers in HistorySummariesMatchRecursion predicate.

(8) 1. ll1HistorySummary ∈ HashType
    BY (7) 3
(8) 2. LL2NVRAMLLogicalHistorySummary' ∈ HistorySummaryType
    BY (2) 1, LL2NVRAMLLogicalHistorySummaryTypeSafe
(8) 3. LL2NVRAM . hashBarrier' ∈ HashType
    BY (2) 1, LL2SubtypeImplicationLemmaDEF LL2SubtypeImplication
(8) 4. HistorySummariesMatchRecursion(  
        ll1HistorySummary, LL2NVRAMLLogicalHistorySummary', LL2NVRAM . hashBarrier' )
    BY (7) 6, (7) 7
(8) 5. QED
    BY (8) 1, (8) 2, (8) 4 DEF HistorySummariesMatchRecursion

One of the conjuncts in the definition of HistorySummariesMatchRecursion is that the Memoir-Opt history summary is a successor of a previous history summary.

(7) 9. LL2HistorySummaryIsSuccessor(  
        LL2NVRAMLLogicalHistorySummary',
        previousLL2HistorySummary,
        prevInput,
        LL2NVRAM . hashBarrier' )
    BY (7) 8 DEF HistorySummariesMatchRecursion

We re-state the definitions from the LET in LL2HistorySummaryIsSuccessor.

(7) successorHistorySummary ≡ Successor(previousLL2HistorySummary, prevInput, LL2NVRAM . hashBarrier')
(7) checkpointedSuccessorHistorySummary ≡ Checkpoint(successorHistorySummary)

We hide the definitions.

(7) HIDE DEF successorHistorySummary, checkpointedSuccessorHistorySummary

373
The definition of $LL2HistorySummaryIsSuccessor$ tells us that there are two ways that the logical history summary could be a successor. We will prove that neither of these disjuncts is satisfiable.

\[
\begin{align*}
\text{(7)10. } & \quad LL2NVRAMLLogicalHistorySummary' = successorHistorySummary \\
& \quad \lor \quad LL2NVRAMLLogicalHistorySummary' = checkpointedSuccessorHistorySummary \\
\end{align*}
\]

BY (7)9
DEF $LL2HistorySummaryIsSuccessor$, $successorHistorySummary$, $checkpointedSuccessorHistorySummary$

First, we prove that the logical history summary cannot be a successor.

\[
\begin{align*}
\text{(7)11. } & \quad LL2NVRAMLLogicalHistorySummary' \neq successorHistorySummary \\
\end{align*}
\]

We re-state a definition from the let in the Successor operator.

\[
\begin{align*}
\text{(8) } & \quad \text{securedInput } \triangleq \text{Hash}(LL2NVRAM.hashBarrier', prevInput) \\
\end{align*}
\]

We hide the definition.

\[
\begin{align*}
\text{(8) } & \quad \text{HIDE DEF securedInput} \\
\end{align*}
\]

There is only one sub-step, which is proving that the extension fields of these two records are unequal. We'll separately prove the two cases of whether or not the primed $SPCR$ equals the base hash value.

\[
\begin{align*}
\text{(8)1. } & \quad LL2NVRAMLLogicalHistorySummary, extension' \neq successorHistorySummary, extension \\
\end{align*}
\]

One fact that will be useful for both cases is that an extension is in progress in the primed state.

\[
\begin{align*}
\text{(9)1. } & \quad LL2NVRAM.extensionInProgess' \\
\end{align*}
\]

\[
\begin{align*}
\text{(10)1. } & \quad LL2NVRAM.extensionInProgress \\
\end{align*}
\]

BY (3)3

\[
\begin{align*}
\text{(10)2. } & \quad \text{UNCHANGED LL2NVRAM.extensionInProgress} \\
\end{align*}
\]

BY (3)2

\[
\begin{align*}
\text{(10)3. } & \quad \text{QED} \\
\end{align*}
\]

BY (10)1, (10)2

The first case is when the $SPCR$ equals the base hash value.

\[
\begin{align*}
\text{(9)2. } & \quad \text{CASE LL2SPCR'} = \text{BaseHashValue} \\
\end{align*}
\]

If an extension is in progress but the $SPCR$ equals the base hash value, the logical history summary equals a crazy hash value, because this situation should never arise during normal operation.

\[
\begin{align*}
\text{(10)1. } & \quad LL2NVRAMLLogicalHistorySummary, extension' = \text{CrazyHashValue} \\
\end{align*}
\]

\[
\begin{align*}
\text{(11)1. } & \quad LL2NVRAMLLogicalHistorySummary' = [ \\
\text{anchor } \mapsto LL2NVRAM.historySummaryAnchor', \\
\text{extension } \mapsto \text{CrazyHashValue}] \\
\end{align*}
\]

BY (9)1, (9)2 DEF $LL2NVRAMLLogicalHistorySummary$

\[
\begin{align*}
\text{(11)2. } & \quad \text{QED} \\
\end{align*}
\]

BY (11)1

The extension field of the successor history summary is equal to a hash generated by the hash function.

\[
\begin{align*}
\text{(10)2. } & \quad successorHistorySummary, extension = \\
\end{align*}
\]

\[
\begin{align*}
\text{Hash}(\text{previousLL2HistorySummary, extension, securedInput}) \\
\end{align*}
\]

BY DEF $successorHistorySummary$, $Successor$, $securedInput$

The arguments to the hash function are both in the hash domain.

\[
\begin{align*}
\text{(10)3. } & \quad \text{previousLL2HistorySummary, extension } \in \text{HashDomain} \\
\end{align*}
\]

\[
\begin{align*}
\text{(11)1. } & \quad \text{previousLL2HistorySummary, extension } \in \text{HashType} \\
\end{align*}
\]

\[
\begin{align*}
\text{(12)1. } & \quad \text{previousLL2HistorySummary } \in \text{HistorySummaryType} \\
\end{align*}
\]

BY (7)8

\[
\begin{align*}
\text{(12)2. } & \quad \text{QED} \\
\end{align*}
\]

BY (12)1 DEF $HistorySummaryType$

\[
\begin{align*}
\text{(11)2. } & \quad \text{QED} \\
\end{align*}
\]

BY (11)1 DEF $HashDomain$

\[
\begin{align*}
\text{(10)4. } & \quad \text{securedInput } \in \text{HashDomain} \\
\end{align*}
\]

\[
\begin{align*}
\text{(11)1. } & \quad \text{securedInput } \in \text{HashType} \\
\end{align*}
\]

\[
\begin{align*}
\text{(11)2. } & \quad \text{QED} \\
\end{align*}
\]

BY (11)1 DEF $HashDomain$
The crazy hash value is not equal to any hash value that can be generated by the hash function when operating on arguments within its domain.

We will prove the second case in two steps.

First, we prove that the extension field of the logical history summary is a hash whose second argument is the fakeHash from the LL2CorruptSPCR action. This follows from the definitions of LL2CorruptSPCR and LL2NVRAMLogicalHistorySummary.

Second, we prove that the extension field of the successor history summary from LL2HistorySummaryIsSuccessor is a hash whose second argument is the secured input from the Successor operator. This follows directly from the definition of successorHistorySummary.

Third, we prove that the two hashes are unequal. We will use the HashCollisionResistant property, along with the fact (which we will prove in a sub-step) that the second arguments to the two hash functions are unequal.

To employ the HashCollisionResistant property, we need to prove some types.
By \{12\} 1 def HashDomain

(11) 2. \( \text{fakeHash} \in \text{HashDomain} \)

By \{3\} 2

(11) 3. \( \text{previousLL2HistorySummary} . \text{extension} \in \text{HashDomain} \)

\( \{12\} 1. \ \text{previousLL2HistorySummary} . \text{extension} \in \text{HashType} \)

\( \{13\} 1. \ \text{previousLL2HistorySummary} \in \text{HistorySummaryType} \)

By \{7\} 8

\( \{13\} 2. \ \text{QED} \)

By \{13\} 1 def HistorySummaryType

\( \{12\} 2. \ \text{QED} \)

By \{12\} 1 def HashDomain

(11) 4. \( \text{securedInput} \in \text{HashDomain} \)

\( \{12\} 1. \ \text{securedInput} \in \text{HashType} \)

\( \{13\} 1. \ \text{LL2NVRAM} . \text{hash Barrier'} \in \text{HashDomain} \)

\( \{14\} 1. \ \text{LL2NVRAM} . \text{hash Barrier'} \in \text{HashType} \)

By \{2\} 1, LL2SubtypeImplicationLemma def LL2SubtypeImplication

\( \{14\} 2. \ \text{QED} \)

By \{14\} 1 def HashDomain

\( \{13\} 2. \ \text{prevInput} \in \text{HashDomain} \)

\( \{14\} 1. \ \text{prevInput} \in \text{InputType} \)

By \{7\} 8

\( \{14\} 2. \ \text{QED} \)

By \{14\} 1 def HashDomain

\( \{13\} 3. \ \text{QED} \)

By \{13\} 1, \{13\} 2, HashTypeSafe def securedInput

\( \{12\} 2. \ \text{QED} \)

By \{12\} 1 def HashDomain

Then we need to prove that the second arguments to the hash functions are unequal. We will employ the restriction on fake hash values from the definition of \( \text{LL2CorruptSPCR} \).

(11) 5. \( \text{fakeHash} \neq \text{securedInput} \)

\( \{12\} 1. \ \forall \text{fakeInput} \in \text{InputType} : \)

\( \begin{align*}
\text{fakeHash} & \neq \text{Hash (LL2NVRAM} . \text{hash Barrier', fakeInput)} \\
\{13\} 1. \ \forall \text{fakeInput} \in \text{InputType} : \\
\text{fakeHash} & \neq \text{Hash (LL2NVRAM} . \text{hash Barrier', fakeInput)} \\
\text{By} \ {3} 2
\end{align*} \)

\( \{13\} 2. \ \text{UNCHANGED LL2NVRAM} . \text{hash Barrier} \)

By \{3\} 2

\( \{13\} 3. \ \text{QED} \)

By \{13\} 1, \{13\} 2

\( \{12\} 2. \ \text{prevInput} \in \text{InputType} \)

By \{7\} 8

\( \{12\} 3. \ \text{QED} \)

By \{12\} 1, \{12\} 2 def securedInput

(11) 6. \text{QED}

Ideally, this \text{QED} step should just read:

By \{11\} 1, \{11\} 2, \{11\} 3, \{11\} 4, \{11\} 5, HashCollisionResistant

However, the prover seems to get a little confused in this instance. We make life easier for the prover by defining some local variables and hiding their definitions before appealing to the HashCollisionResistant assumption.

\( \{12\} \ h1a \overset{\triangle}{=} \text{LL2SPCR} \)

\( \{12\} \ h2a \overset{\triangle}{=} \text{fakeHash} \)

376
\( h_1b \triangleq \text{previousLL}_2\text{HistorySummary}\.\text{extension} \)
\( h_2b \triangleq \text{securedInput} \)
\( h_1a \in \text{HashDomain} \)
\( h_2a \in \text{HashDomain} \)
\( h_1b \in \text{HashDomain} \)
\( h_2b \in \text{HashDomain} \)
\( h_2a \neq h_2b \)
\( \text{Hash}(h_1a, h_2a) \neq \text{Hash}(h_1b, h_2b) \)
\( \text{Extended} \text{Definition} h_1a, h_2a, h_1b, h_2b \)
\( \text{HashCollisionResistant} \)

The two cases are exhaustive.

Second, we prove that the logical history summary cannot be a checkpoint.

\( \text{Extended} \text{Definition} \text{LL}_2\text{NVRAMLogicalHistorySummary}' \neq \text{checkpointedSuccessorHistorySummary} \)

The extension field of the logical history summary does not equal the base hash value, as we proved above.

\( \text{CheckpointHasBaseExtensionLemma} \)

\( \text{Extended} \text{Definition} \text{LL}_2\text{NVRAMLogicalHistorySummary}.\text{extension}' \neq \text{BaseHashValue} \)

The extension field of the checkpointed successor does equal the base hash value. This follows from the \( \text{CheckpointHasBaseExtensionLemma} \).

\( \text{Extended} \text{Definition} \text{LL}_2\text{NVRAMLogicalHistorySummary}.\text{extension} = \text{BaseHashValue} \)

We thus have a contradiction.
We prove that the constraint labeled previous in the LL1RestrictedCorruption action is satisfied.

To prove the universally quantified expression, we take a set of variables of the appropriate types.

We re-state the definition from within the LL1RestrictedCorruption!nvram!previous clause.

We hide the definition.

We need to prove the nvram::previous conjunct, which asserts an implication. It suffices to assume the antecedent and prove the consequent.

The consequent of the nvram::previous conjunct asserts that the authenticator is not a valid MAC for the history state binding formed from any predecessor of the history summary in the NVRAM and any state hash.

We will use proof by contradiction.

We first pick, from the set of Memoir-Opt observed authenticators, a Memoir-Opt authenticator that matches the Memoir-Basic authenticator. We know that such a authenticator exists, because the refinement asserts that the sets of observed authenticators match across the two specs.
We prove that the Memoir-Basic’s history summary picked to satisfy the AuthenticatorsMatch predicate.

We pick a set of variables of the appropriate types that satisfy the quantified AuthenticatorsMatch predicate.

We re-state the definitions from the let in AuthenticatorsMatch.

We prove the types of the definitions, with help from the AuthenticatorsMatchDefsTypeSafeLemma.

We hide the definitions.

We prove that the Memoir-Basic’s history summary picked to satisfy the AuthenticatorsMatch predicate equals the history summary in the primed state of the Memoir-Basic NVRAM.

The first step is to show the equality of the history state bindings that bind each of these history summaries to their respective state hashes.

By hypothesis, the authenticator is a valid MAC for the garbage history state binding.

The definition of the AuthenticatorsMatch predicate tells us that the Memoir-Basic authenticator was generated as a MAC from the history state binding.

The remaining preconditions are types.
SomeHistoryStateBinding ∈ HashType

SomeHistorySummary ∈ HashDomain

SomeHistorySummary ∈ HashType

The MACCollisionResistant property tells us that the two history state bindings are equal.

By the collision resistance of the hash function, the equality of the history state bindings implies the equality of the history summaries.

We pick a value for the Memoir-Opt previous-inputs-summary existential variable inside the HistorySummariesMatchRecursion predicate that satisfies this predicate for (1) the Memoir-Basic's history summary picked to satisfy the AuthenticatorsMatch predicate and (2) the input taken from the universal quantifier in the previous conjunct in the LL1RestrictedCorruption action.
The Memoir’s history summary picked to satisfy the AuthenticatorsMatch predicate matches the primed logical history summary in the Memoir-Opt NVRAM and SPCR, by the refinement and the above equality.

(8.1) HistorySummariesMatch(
    LL1NVRAM.historySummary',
    LL2NVRAMLLogicalHistorySummary',
    LL2NVRAM.hashBarrier')

(9.1) HistorySummariesMatch(
    LL1NVRAM.historySummary',
    LL2NVRAMLLogicalHistorySummary',
    LL2NVRAM.hashBarrier')

BY (2)1 def LL2Refinement

(9)2. QED

BY (9)1

We prove that the HistorySummariesMatch predicate equals the HistorySummariesMatchRecursion predicate in this case. We assert each condition required by the definition of the predicate.

(8)2. HistorySummariesMatch(
    LL1NVRAM.historySummary',
    LL2NVRAMLLogicalHistorySummary',
    LL2NVRAM.hashBarrier') =
    HistorySummariesMatchRecursion(
    LL1NVRAM.historySummary',
    LL2NVRAMLLogicalHistorySummary',
    LL2NVRAM.hashBarrier')

We prove some types, to satisfy the universal quantifiers in HistorySummariesMatchDefinition.

(9)1. LL1NVRAM.historySummary' ∈ HashType
BY (2)1 def LL2Refinement, LL1TrustedStorageType

(9)2. LL2NVRAMLLogicalHistorySummary' ∈ HistorySummaryType
BY (2)1, LL2NVRAMLLogicalHistorySummaryTypeSafe

(9)3. LL2NVRAM.hashBarrier' ∈ HashType
BY (2)1, LL2SubtypeImplicationLemma

We prove that the Memoir-Opt logical history summary does not equal the initial history summary.

(9) ll2InitialHistorySummary ≡ [anchor → BaseHashValue, extension → BaseHashValue]

(9)4. LL2NVRAMLLogicalHistorySummary' ≠ ll2InitialHistorySummary
(10)1. LL2NVRAMLLogicalHistorySummary'.extension' ≠ BaseHashValue
BY (4)1

(10)2. ll2InitialHistorySummary.extension = BaseHashValue
BY DEF ll2InitialHistorySummary

(10)3. QED
BY (10)1, (10)2

Finally, from HistorySummariesMatchDefinition, we can conclude that the HistorySummariesMatch predicate equals the quantified HistorySummariesMatchRecursion predicate.

(9)5. QED
BY (9)1, (9)2, (9)3, (9)4, HistorySummariesMatchDefinition
DEF ll2InitialHistorySummary

We pick values for the remaining two existential variables inside the HistorySummariesMatchRecursion predicate that satisfy the predicate. We know such variables exist, because the predicate is satisfied by the two previous steps.

(8)3. PICK prevInput ∈ InputType,
previousLL1HistorySummary ∈ HashType,
previousLL2HistorySummary ∈ HistorySummaryType :
HistorySummariesMatchRecursion(
    LL1NVRAM.historySummary',
    LL2NVRAML0gicalHistorySummary',
    LL2NVRAM.hashBarrier')
    (prevInput,
    previousLL1HistorySummary,
    previousLL2HistorySummary)

We prove some types, to satisfy the universal quantifiers in HistorySummariesMatchRecursion predicate.

(9.1) ll1HistorySummary ∈ HashType
    BY (7)3

(9.2) LL2NVRAMLogicalHistorySummary' ∈ HistorySummaryType
    BY (2)1, LL2NVRAML0gicalHistorySummaryTypeSafe

(9.3) LL2NVRAM.hashBarrier' ∈ HashType
    BY (2)1, LL2SubtypeImplicationLemma

(9.4) HistorySummariesMatchRecursion(
    LL1NVRAM.historySummary',
    LL2NVRAMLogicalHistorySummary',
    LL2NVRAM.hashBarrier')
    BY (8)1, (8)2

(9.5) QED
    BY (9)1, (9)2, (9)4 DEF HistorySummariesMatchRecursion

We prove that the existential variables for the previous input and the previous history summary in the above pick are equal to the input and history summary taken from the universal quantifiers in the previous conjunct in the LL1RestrictedCorruption action. We use the HashCollisionResistant property.

(8.4) ∧ ll1SomeHistorySummary = previousLL1HistorySummary
    ∧ someInput = prevInput

We prove the necessary types for the HashCollisionResistant property.

(9.1) ll1SomeHistorySummary ∈ HashDomain
    (10)1. ll1SomeHistorySummary ∈ HashType
            BY (6)1
    (10)2. QED
            BY (10)1 DEF HashDomain

(9.2) someInput ∈ HashDomain
    (10)1. someInput ∈ InputType
            BY (6)1
    (10)2. QED
            BY (10)1 DEF HashDomain

(9.3) previousLL1HistorySummary ∈ HashDomain
    (10)1. previousLL1HistorySummary ∈ HashType
            BY (8)3
    (10)2. QED
            BY (10)1 DEF HashDomain

(9.4) prevInput ∈ HashDomain
    (10)1. prevInput ∈ InputType
            BY (8)3
    (10)2. QED
            BY (10)1 DEF HashDomain

The hashes are equal, because each is equal to the history summary in the primed Memoir-Basic NVRAM.

(9.5) Hash(ll1SomeHistorySummary, someInput) =
    Hash(previousLL1HistorySummary, prevInput)
The hash of the taken history summary and input are equal to the history summary in the primed Memoir-Basic NVRAM by assumption of the antecedent in the previous conjunct in the L1RestrictedCorruption action.

(10)1. \( L1NVRAM.historySummary' = Hash(ll1SomeHistorySummary, someInput) \)
   \[ \text{by (6)2} \]

The hash of the picked history summary and input are equal to the history summary in the primed Memoir-Basic NVRAM by the definition of the HistorySummariesMatchRecursion predicate.

(10)2. \( L1NVRAM.historySummary' = Hash(previousLL1HistorySummary, prevInput) \)
   \[ \text{by (8)3} \]
(10)3. QED
   \[ \text{by (10)1, (10)2} \]
(9)6. QED
   \[ \text{by (9)1, (9)2, (9)3, (9)4, (9)5, HashCollisionResistant} \]
(8)5. QED
   \[ \text{by (8)3, (8)4} \]

One of the conjuncts in the definition of HistorySummariesMatchRecursion is that the Memoir-Opt history summary is a successor of a previous history summary.

(7)7. \( LL2HistorySummaryIsSuccessor(\)
   \[ \text{LL2NVRAMLogicalHistorySummary'}, \]
   \[ \text{previousLL2HistorySummary,} \]
   \[ \text{someInput,} \]
   \[ \text{LL2NVRAM.hashBarrier')} \]
   \[ \text{by (7)6 DEF HistorySummariesMatchRecursion} \]

We re-state the definitions from the LET in LL2HistorySummaryIsSuccessor.

(7) successorHistorySummary \( \triangleq \) Successor(previousLL2HistorySummary, someInput, LL2NVRAM.hashBarrier')
(7) checkpointedSuccessorHistorySummary \( \triangleq \) Checkpoint(successorHistorySummary)

We hide the definitions.

(7) HIDE DEF successorHistorySummary, checkpointedSuccessorHistorySummary

The definition of LL2HistorySummaryIsSuccessor tells us that there are two ways that the logical history summary could be a successor. We will prove that neither of these disjuncts is satisfiable.

(7)8. \( \lor \) LL2NVRAMLogicalHistorySummary' = successorHistorySummary
   \[ \lor \] LL2NVRAMLogicalHistorySummary' = checkpointedSuccessorHistorySummary
   \[ \text{by (7)7} \]
   \[ \text{DEF LL2HistorySummaryIsSuccessor, successorHistorySummary,} \]
   \[ \text{checkpointedSuccessorHistorySummary} \]

First, we prove that the logical history summary cannot be a successor.

(7)9. LL2NVRAMLogicalHistorySummary' \( \neq \) successorHistorySummary

We re-state a definition from the LET in the Successor operator.

(8) securedInput \( \triangleq \) Hash(LL2NVRAM.hashBarrier', someInput)

We hide the definition.

(8) HIDE DEF securedInput

There is only one sub-step, which is proving that the extension fields of these two records are unequal.

(8)1. LL2NVRAMLogicalHistorySummary'.extension' \( \neq \) successorHistorySummary.extension

First, we prove that the extension field of the logical history summary is a hash whose second argument is the fakeHash from the LL2CorruptSPCR action. This follows from the definitions of LL2CorruptSPCR and LL2NVRAMLogicalHistorySummary.

(9)1. LL2NVRAMLogicalHistorySummary'.extension' = Hash(LL2SPCR, fakeHash)
(10)1. LL2SPCR' = Hash(LL2SPCR, fakeHash)
   \[ \text{by (3)2} \]
Second, we prove that the extension field of the successor history summary from \( LL2HistorySummaryIsSuccessor \) is a hash whose second argument is the secured input from the \( Successor \) operator. This follows directly from the definition of \( successorHistorySummary \).

\[
\text{(9)2.} \quad \text{\texttt{successorHistorySummary.extension} =}
\]
\[
\text{\texttt{Hash(previousLL2HistorySummary.extension, securedInput)}}
\]

\[
\text{\texttt{BY DEF successorHistorySummary, Successor, securedInput}}
\]

Third, we prove that the two hashes are unequal. We will use the \( HashCollisionResistant \) property, along with the fact (which we will prove in a sub-step) that the second arguments to the two hash functions are unequal.

\[
\text{(9)3.} \quad \text{\texttt{Hash(LL2SPCR, fakeHash) \neq}}
\]
\[
\text{\texttt{Hash(previousLL2HistorySummary.extension, securedInput)}}
\]

\[
\text{\texttt{To employ the HashCollisionResistant property, we need to prove some types.}}
\]

\[
\text{(10)1.} \quad \text{\texttt{LL2SPCR} \in HashDomain}
\]
\[
\text{\texttt{BY (2)1 DEF LL2TypeInvariant}}
\]
\[
\text{\texttt{BY (11)1 DEF HashDomain}}
\]
\[
\text{\texttt{fakeHash} \in HashDomain}
\]
\[
\text{\texttt{BY (3)2}}
\]
\[
\text{\texttt{previousLL2HistorySummary.extension} \in HashDomain}
\]
\[
\text{\texttt{BY (11)1 DEF HashDomain}}
\]
\[
\text{\texttt{previousLL2HistorySummary.extension} \in HashType}
\]
\[
\text{\texttt{BY (12)1 DEF HistorySummaryType}}
\]
(11) 2. QED
   BY (11) 1 DEF HashDomain

(10) 4. securedInput ∈ HashDomain

(11) 1. securedInput ∈ HashType
   ⟨12⟩ 1. LL2NVRAM.hashBarrier’ ∈ HashDomain
   ⟨13⟩ 1. LL2NVRAM.hashBarrier’ ∈ HashType
      BY (2) 1, LL2SubtypeImplicationLemma DEF LL2SubtypeImplication
   ⟨13⟩ 2. QED
      BY (13) 1 DEF HashDomain
   ⟨12⟩ 2. someInput ∈ HashDomain
   ⟨13⟩ 1. someInput ∈ InputType
      BY (6) 1
   ⟨13⟩ 2. QED
      BY (13) 1 DEF HashDomain

(12) 3. QED
      BY (12) 1, (12) 2, HashTypeSafe DEF securedInput

(11) 2. QED
      BY (11) 1 DEF HashDomain
Then we need to prove that the second arguments to the hash functions are unequal. We will employ
the restriction on fake hash values from the definition of LL2CorruptSPCR.

(10) 5. fakeHash ≠ securedInput
(11) 1. ∀ fakeInput ∈ InputType :
    fakeHash ≠ Hash(LL2NVRAM.hashBarrier’, fakeInput)
   ⟨12⟩ 1. ∀ fakeInput ∈ InputType :
      fakeHash ≠ Hash(LL2NVRAM.hashBarrier, fakeInput)
      BY (3) 2
   ⟨12⟩ 2. UNCHANGED LL2NVRAM.hashBarrier
      BY (3) 2
   ⟨12⟩ 3. QED
      BY (12) 1, (12) 2
(11) 2. someInput ∈ InputType
      BY (6) 1
(11) 3. QED
      BY (11) 1, (11) 2 DEF securedInput

(10) 6. QED

Ideally, this QED step should just read:

BY (10) 1, (10) 2, (10) 3, (10) 4, (10) 5, HashCollisionResistant

However, the prover seems to get a little confused in this instance. We make life easier for the
prover by defining some local variables and hiding their definitions before appealing to the
HashCollisionResistant assumption.

(11) h1a ≜ LL2SPCR
(11) h2a ≜ fakeHash
(11) h1b ≜ previousLL2HistorySummary.extension
(11) h2b ≜ securedInput
(11) 1. h1a ∈ HashDomain
      BY (10) 1
(11) 2. h2a ∈ HashDomain
      BY (10) 2
(11) 3. h1b ∈ HashDomain
      BY (10) 3
(11) 4. h2b ∈ HashDomain

385
Second, we prove that the logical history summary cannot be a checkpoint.

(7)10. \( \text{LL}2\text{NVRAM}\text{LogicalHistorySummary} \neq \text{checkpointedSuccessorHistorySummary} \)

The extension field of the logical history summary does not equal the base hash value, as we proved above.

(8)1. \( \text{LL}2\text{NVRAM}\text{LogicalHistorySummary}.\text{extension} \neq \text{BaseHashValue} \)

The extension field of the checkpointed successor does equal the base hash value. This follows from the \text{CheckpointHasBaseExtensionLemma}.

(9)1. \( \text{successorHistorySummary} \in \text{HistorySummaryType} \)

(10)1. \( \text{previousLL}2\text{HistorySummary} \in \text{HistorySummaryType} \)

(10)2. \( \text{someInput} \in \text{InputType} \)

(10)3. \( \text{LL}2\text{NVRAM}.\text{hashBarrier} \in \text{HashType} \)

(10)4. \( \text{QED} \)

(9)2. \( \text{QED} \)

We thus have a contradiction.

(7)11. \( \text{QED} \)

We prove the third conjunct within the \text{nram} conjunct of the \text{LL1RestrictedCorruption} action.

(5)4. \( \text{LL}1\text{NVRAM}' = [ \)

\[ \text{historySummary} \mapsto \text{LL}1\text{NVRAM}.\text{historySummary}', \]

\[ \text{symmetricKey} \mapsto \text{LL}1\text{NVRAM}.\text{symmetricKey} \] \)

(6)1. \( \text{LL}1\text{NVRAM} \in \text{LL1TrustedStorageType} \)

(6)2. \( \text{LL}1\text{NVRAM}' \in \text{LL1TrustedStorageType} \)

(6)3. \( \text{UNCHANGED LL}1\text{NVRAM}.\text{symmetricKey} \)

(7)1. \( \text{UNCHANGED LL}2\text{NVRAM}.\text{symmetricKey} \)

(3)1. \( \text{LL2CorruptSPCR} \)
We first prove that the Memoir-Basic NVRAM. The remaining variables are unchanged, so we can address each with its appropriate lemma.

Next, we prove the conjunct relating to the RAM. For \( LL2CorruptSPCR \), the RAM is unchanged, so we can simply invoke the UnchangedRAMLemma.

The remaining variables are unchanged, so we can address each with its appropriate lemma.

For the ELSE case, we assume that an extension is not in progress and show that this refines to a stuttering step.

We first prove that the Memoir-Basic NVRAM is unchanged. We do this one field at a time.

The history summary in the Memoir-Basic NVRAM is unchanged.

The logical history summary in the Memoir-Opt NVRAM and SPCR is unchanged.

We reveal the definition of the unprimed logical history summary in the Memoir-Opt NVRAM and SPCR, given that there is no extension in progress.

We reveal the definition of the primed logical history summary in the Memoir-Opt NVRAM and SPCR, given that there is no extension in progress.
Both then and else cases are proven.

(3)5. QED
BY (3)3, (3)4

(2)12. QED

BY (2)1, (2)2, (2)3, (2)4, (2)5, (2)6, (2)7, (2)8, (2)9, (2)10, (2)11 DEF LL1Next, LL2Next

(1)4. QED

Using the StepSimulation proof rule, the base case and the induction step together imply that the implication always holds.

(2)1. □[LL2Next]LL2Vars ∧ □LL2Refinement ∧ □LL2TypeInvariant ⇒ □[LL1Next]LL1Vars

BY (1)3, StepSimulation

(2)2. QED

BY (1)1, (1)2, (2)1 DEF LL2Spec, LL1Spec, LL2Refinement
ACKNOWLEDGMENTS
The authors are deeply in debt to Leslie Lamport, not only for creating the TLA+ language, refinement-based proof methodology, and hierarchical proof structure, but also for his ongoing tutelage and ready assistance during the long process of writing the proofs herein. We also thank Jon Howell for early discussions and advice on the best approach for proving correctness of the Memoir system.

REFERENCES