1 Introduction

Once a formal specification for a system is written, it is usually desirable to check that the specification conforms to some concept of correctness. One way to do this is to prove that the specification is a refinement of a simpler specification whose correctness is more obvious. This refinement approach is used in two stages in proving that the Memoir system is correct [9, 3]. Another way is to write invariants, or safety properties, and then directly prove that the specification maintains the invariants.

Of course, in either case there could be oversights in the proof, so to be certain that the proof is correct, it must be a formal proof checked by a mechanical proof checker. Unfortunately, such formal proofs tend to be lengthy and tedious, since the limited deductive power of current mechanical proof checkers requires detailed proof steps. So it is a good idea to be fairly confident that the specification is correct before investing the effort in writing a formal proof.

To get confidence that a specification maintains its invariants, one can use a model checker to explore the state space. Unfortunately, a model checker is limited to exploring a finite number of states, and the state space usually explodes rapidly as the model configuration parameters are increased. Although model checking may have found no errors in the specification, there is always the question of whether, if there were an error, would the model checker have been able to find it within the configurations that are feasible to check? To address this question, we propose introducing some errors on purpose and seeing if the model checker can find them.

We present the results of model checking, with inserted errors, a TLA+ [7] specification for a node in Pasture, a
messaging library that provides secure off-line access to data using a TPM. The state space of even small configurations of the Pasture specification is too large to gain much confidence by direct model checking, but observing that intentional bugs can be found within the configurations that can be checked gives a reasonable confidence that the specification is correct. We then go on to describe a formal proof that the specification maintains its invariants. The formal proof has been checked using the TLA+ Proof System [2].

After writing, checking, and proving the correctness of the Pasture specification, we realized that the implementation could be optimized by combining the functions of two of Pasture’s Program Configuration Registers (PCRs) into one PCR. Surprisingly enough, it turned out to be a simple matter to edit our initial formal specification and proof to produce a formal specification and proof of optimized Pasture.

2 Overview of Pasture

Pasture [5] is a messaging library that provides secure off-line access to data. When on-line, the receiver downloads an encrypted copy of the data from a sender. Later, when off-line, the receiver makes a decision either (1) to obtain access to the decryption key and thus to the data, or (2) to revoke access to the decryption key and thus effectively delete the data without reading it.

Pasture provides two safety properties: access undeniability and verifiable revocation. Access undeniability means that a receiver cannot deny any decision it made to obtain access to the data and still survive an audit. Verifiable revocation means that a receiver can provide a proof of revocation for any decision it made to revoke access to the data. This proof establishes that the receiver never did and never will be able to access that data.1

These properties could be used, for example, by a video rental service. The receiver could pay for and download an encrypted video from the sender. Later, the receiver could decide whether to obtain access to the video and watch it, or revoke access and never watch it. Afterwards, if access was revoked, the receiver could present the proof of revocation to the sender and get a refund.

Pasture works by implementing a tamper-evident append-only log of decisions on the receiver. Figure 1 shows the protocol.2 For this paper we concentrate on the implementation of the tamper-evident append-only log and how it relates to the use of a decryption key and the production of a proof of revocation.

Pasture uses a Trusted Platform Module (TPM) [12, 1] in the receiver to maintain a cryptographic summary of the receiver’s log and to protect decryption keys. We assume that the reader is generally familiar with how TPMs work.

The cryptographic summary of the receiver’s log is maintained in a Platform Configuration Register (PCR) inside the receiver’s TPM. A PCR can be updated only via the TPM primitive TPM_Extend, which corresponds to the action of appending a value (called a measurement in the TPM literature) to the log. A given PCR value serves as a cryptographically unique representation of the sequence of measurements used to produce it, since it is

---

1. The properties apply only when the sender is correct. A faulty sender could just send the data in the clear to a receiver, and there could be no guarantee about whether the receiver accessed the data or not. The intent is to protect a correct sender against a faulty receiver.

2. Certain details related to preventing spoofing have been omitted, such as signatures on the messages. Also, we omit showing that the sender should verify the proof KP before encrypting the message.
cryptographically impossible to determine any other sequence that would produce the same result. Pasture uses PCR

The TPM primitive TPM_CreateWrapKey is used to bind the decryption key to a potential future state of the log, in which the current log has been extended by the decision to obtain access to the key. This decision is represented in the log as the cryptographic hash $hm$ of the message. To revoke access to the key, the receiver instead extends its log by $\delta \neq hm$. Assuming that the cryptography is unbreakable, this extension makes it impossible to reach the log state to which the decryption key is bound, and thus makes it impossible ever to use the decryption key.

Since the TPM’s PCRs are volatile, and are reset to their initial values on reboot, the main difficulty faced by Pasture is how to preserve its state across reboots. If an adversary could rollback Pasture state to an earlier point, arranging to violate Pasture’s safety properties of access undeniability and verifiable revocation would be easy.

Memoir [9] presented a solution to this problem for any deterministic application. Memoir maintains a cryptographic log summary of application states ub a PCR much like Pasture’s PCRApp. The optimized Memoir solution adds a checkpoint routine to the system shutdown sequence and a recovery routine to the system boot sequence. The checkpoint routine copies the PCR to an NVRAM location and then sets an NVRAM flag indicating that the copy is current. The recovery routine checks that the NVRAM value is marked as current and if so plays back measurements from the full log, re-extending the PCR until its contents matches the value saved in the NVRAM. Memoir uses an ExtensionSecret to prevent an adversary from duplicating a prefix of the re-extension process. Any time that the log is extended, when the measurement is extended on the PCR the NVRAM flag is cleared indicating that the NVRAM copy of the PCR is no longer current.

Memoir exploits Secure Execution Mode (SEM) in the manner developed by Flicker [8]. SEM enables a routine to run in a protected environment, with interrupts, other cores, and DMA disabled, and with a special PCRSEM set to a value (otherwise cryptographically unreachable) based on a cryptographic hash of the routine.

Pasture adopts most of the Memoir approach, with a few modifications so that the normal Pasture operations of

**Figure 2: Pasture operations.**
CreateBoundKey, ObtainAccess, and RevokeAccess do not need to run in SEM. In this way Pasture exploits the specific nature of its application to obtain a solution with much less overhead in its particular case. Figure 2 shows the implementation of Pasture operations.

Pasture’s Recover operation re-extends PCR\textsubscript{APP} from the full log, then enters SEM to verify that PCR\textsubscript{APP} matches the value saved in the NVRAM (and that the value in NVRAM is current). If so, PCR\textsubscript{SEM} is extended by Happy, to produce a value SemHappy that can be reached in no other way, and the NVRAM flag is cleared to indicate that the NVRAM value is no longer current. Otherwise, PCR\textsubscript{SEM} is extended by Unhappy, producing a different value.

CreateBoundKey requires both that PCR\textsubscript{APP} contain the proposed future log summary and that PCR\textsubscript{SEM} contain SemHappy in order for decryption to be possible. The adversary could reboot the system and re-extension PCR\textsubscript{APP}, but cannot arrange for PCR\textsubscript{SEM} to contain SemHappy and so cannot rollback and access a decryption key.

Likewise, RevokeAccess and Audit quote both PCR\textsubscript{APP} and PCR\textsubscript{SEM} in order to prove that PCR\textsubscript{APP} was quoted at a time when PCR\textsubscript{SEM} contained SemHappy.

Pasture’s Checkpoint operation has a difficulty. It needs to verify that PCR\textsubscript{SEM} contains SemHappy so that it can trust the current contents of PCR\textsubscript{APP}, but it has to enter SEM in order to protect its actions from interference by the adversary. Its solution is to use a transport session SEAL to get an attestation \(\alpha\) of the contents of PCR\textsubscript{APP} and PCR\textsubscript{SEM} before entering SEM. A TPM monotonic counter CTR is used to prevent the adversary from replaying an earlier SEAL attestation.

Taking the SEAL also has to destroy the usefulness of PCR\textsubscript{APP}, or else an adversary could take the SEAL, extend PCR\textsubscript{APP} to obtain access to a key or generate a verifiable proof of revocation, and then pass the SEAL to the checkpoint SEM routine and continue with a normal reboot and recovery, which would rollback the actions that the adversary performed after taking the SEAL. For this purpose, PCR\textsubscript{SEAL} is used. PCR\textsubscript{SEAL} normally contains its initial value SealReboot, which is checked in CreateBoundKey and quoted in RevokeAccess and Audit. The SEAL transport session extends PCR\textsubscript{SEAL} thus rendering PCR\textsubscript{APP} useless until the next reboot.

3 The specification

Appendix A gives a TLA+ [7] specification of the state within a Pasture node. The specification closely tracks the Pasture operations shown in Figure 2 and also models the actions of an adversary who has the power to extend PCRs, to observe whatever attestations are created, to invoke Pasture’s secure execution mode routines with any parameters known to the adversary, and to reboot the node at arbitrary times. Since we assume that cryptography cannot be broken, the adversary does not have the power to forge attestations or to set PCRs to an arbitrary value.

The specification abstracts the Pasture node in the following ways:

- **Only one hash.** The specification models the hash \(\delta\) as just one value, Pcr\textsubscript{OBTAIN}. The revoke measurement \(\delta\) is modeled as Pcr\textsubscript{REVOKE}. Note that if multiple, distinct hash values were modeled, the specification would be symmetric over permutations of the hash values. Modeling all hash values as just the one value Pcr\textsubscript{OBTAIN} eliminates this symmetry from the specification. No descriptive power is lost, because the specification does not admit any actions that compare isolated hash values.

- **Potential key bindings.** The specification assumes that a key may be bound to any current state of the log extended by Pcr\textsubscript{OBTAIN}. Extending the log by Pcr\textsubscript{OBTAIN} obtains access to this key and extending the log by Pcr\textsubscript{REVOKE} revokes access to this key.

- **Recovery.** In Pasture, recovery first re-extends PCR\textsubscript{APP} from measurements recorded in the full log and then enters SEM to verify that the resulting value in PCR\textsubscript{APP} is current. The specification accomplishes the re-extension by allowing any possible sequence of extensions of PCR\textsubscript{APP}, since this is within the power of the adversary. The specification models recovery as the re-extension sequence that happens to be the correct one.

\(^3\)The protection of Pasture’s NVRAM depends on the assumption that the value SemProtect is present in PCR\textsubscript{SEM} only during Pasture’s secure execution mode routines, which is the subject of the invariant InvNvProtection.
• **Checkpoint.** In Pasture, checkpoint first performs a SEAL transport session and gets the attestation, and then enters SEM to verify the attestation and then record the log summary in NVRAM. The specification collects a knowledge of all SEAL attestations that have ever been generated and allows choosing any known one for the SEM routine to verify, since this is within the power of the adversary. The specification models checkpoint as choosing the correct one.

The specification starts with a series of Bug definitions all set to FALSE. Overriding one of these definitions with TRUE introduces a bug into the specification as discussed later in Section 5.

Next the specification introduces definitions for PCRs. A PCR is modeled as an initial value in Pcri combined with a sequence of extensions in Pcrx.

Next the specification introduces definitions for PC values within Secure Execution Mode (SEM). When the node is in SEM the adversary cannot interpose any actions except to reboot the node.

Next the specification introduces definitions for Pasture’s protected NVRAM, the SEAL transport session, and then finally the state of the entire node.

There are two “fiduciary” variables which are used for expressing invariants: obtains and revokes.

The variable obtains is a set that contains all application PCR values that have been used to obtain a key. Since the last decision logged must be the decision to obtain the key, these PCR values all have PcrxOBTAIN as their final extension.

The variable revokes is a set that contains all application PCR values have been used for a proof of revocation. Since the last decision logged must be the decision to revoke a key, these PCR values all have PcrxREVOKE as their final extension.

Next the specification introduces the next state relation decomposed as a long series of actions. Then the actual Init and Next definitions of the specification are presented, followed by the complete specification Spec.

Finally, the specification introduces a list of invariants. The invariant InvType asserts that all variables always contain values of the correct type. The invariant InvNv-Protection asserts that access to Pasture’s NVRAM region (which is controlled by the value contained in the secure execution mode PCR) is permitted precisely when the node is in secure execution mode. The invariants InvAccessUndeniability and InvVerifiableRevocation correspond to the main safety properties of Pasture.

Access undeniability is equivalent to saying that whenever the node is auditable, every element in obtains is a prefix of the current application PCR. This means that whenever a node is auditable, it must provide a full log that lists every decision it made to obtain access to a key.

Verifiable revocation is equivalent to saying that there is no PCR o ∈ obtains and PCR r ∈ revokes such that everything in o except the last decision (which must be OBTAIN) matches everything in r except the last decision (which must be REVOKE). If it were possible to have such an o and r, it would mean that there would be a key for which both access was obtained and also a proof of revocation was generated.

### 4 Model checking

Appendix B shows a TLA+ specification for model checking the Pasture node specification. The model specification creates an instance of PastureNode with constants for the initial values and extensions of PCRs. The constants are carefully chosen to be a minimal set that satisfies the required assumptions.

The model specification also introduces a parameterized constraint to limit the number of states to a finite number. The parameters of the constraint are as follows:

- **MaxAppPcrLen.** The maximum number of extensions of PCRAPP; and therefore the maximum number of entries in the log and the maximum number of keys for which access can be obtained or revoked.

- **MaxSemPcrLen.** The maximum number of extensions of PCRSEM. Pasture requires at least one, so that the Pasture SEM routines can extend PCRSEM before exiting, which is required to remove access privileges from Pasture’s NVRAM. Note that the specification does not count entering a SEM routine as requiring an extension to PCRSEM, but merely initializes PCRSEM with SemProtected which represents the result of resetting and then extending with the cryptographic module hash. The TPM semantics
of resetting PCRSEM ensures that it is cryptographically impossible to reach SemProtected in any other way.

- **MaxSealPcrLen.** The maximum number of extensions of PCRSEAL. Pasture requires at least one, so that the SEAL transport session can extend PCRSEAL.

- **MaxTsValues.** The maximum number of SEAL attestations that can be known at any one time. Pasture requires at least one, so that the most recent SEAL attestation can be provided to the Checkpoint SEM routine. Note that the specification admits of forgetting a SEAL attestation that once was known. This permits model checking a Pasture configuration through multiple reboots with only one SEAL attestation known at a time, since only the most recent one needs to be remembered for Pasture to continue to function.

- **MaxBootCtr.** The maximum value of the boot counter. Pasture increments this counter once each time through the Checkpoint routine.

We used the TLA+ toolbox [6] with TLC2 version 2.05 to model check the specification for various configurations. For each configuration, TLC determined the maximum depth of the state space graph as well as the total number of distinct states. No violations were found. Table 1 shows the results.

For brevity, we refer to a specific configuration by listing the parameter values left-to-right in the order shown in Table 1. For example, the (2,1,2,2,2) configuration is the last configuration listed in the table.

We started by model checking configurations on an Intel Core™ i7 M620 laptop with 4 GB of memory and 4 cores @ 2.67 GHz. As expected, the number of distinct states and consequently the model checking run time increased enormously as the configuration parameters were increased. Figure 3 shows TLC’s agonizing plot of queue size over time for the largest configuration we model checked using the laptop. The rate of next state exploration became particularly slow after about two hours of run time as TLC was completely disk-bound.

To check some larger configurations, we obtained unshared access to an AMD Opteron™ 6168 server with 128 GB of memory and 48 cores @ 1.90 GHz. However, even using this large server machine, the enormous state space explosion of the Pasture node specification exposed some limitations in TLC.

The (2,1,1,2,3) configuration has over 379 million dis-
tinct states, so there is a non-trivial probability of finger-
pint collision. Such a collision would cause TLC to fail
to explore the complete state space. TLC reported a calcu-
lated collision probability of 0.058 and an observed col-
lision probability of 0.027. We re-ran the configuration
with a different fingerprint seed and four hours later were
pleased to see that the second run explored the same num-
ber of distinct states, this time reporting an observed col-
lision probability of 0.002. We performed a third run with
yet another seed, and TLC again explored the same num-
ber of distinct states. At this point we decided that the
TLC runs on this configuration were almost certainly not
suffering from fingerprint collision.

The (2,1,2,2,2) configuration has over 4 billion distinct
states. According to the birthday paradox, the probability
of a 64-bit fingerprint collision among this many states is
0.38. TLC reported an observed collision probability of
1.0, for whatever that is worth. We re-ran the configu-
ration with a different fingerprint seed and two and a half
days later were pleased to see that the second run explored
the same number of distinct states. We performed a third
run with yet another seed, and this time TLC explored five
fewer distinct states. The number of distinct states listed
for this configuration in Table 1 is the number explored in
the first and second runs. However, with these results it
is not clear whether or not TLC is actually exploring the
entire state space. None of the runs found any errors.

Clearly, the probability of a fingerprint collision would
make the results of running TLC on any larger configu-
rations fairly inconclusive, even if we wanted to wait for
such a run to complete.

We did not apply SYMMETRY in our model checking
runs because the specification as written has none. The
use of the one value PcrxOBTAION as a model for any
hash value \( hM \) wrings out the one symmetry that would
be present in a more detailed specification.

It was nice to see that none of the invariants were vio-
lated for the configurations that TLC could check. How-
ever, there is always the possibility that a bug lurks over
the horizon. Normally, we would like to check configu-
rations with parameter values up to at least three. In our
experience, a system will often have interesting behavior
when there is the chance for three instances of something
to interact. But in model checking the Pasture specifi-
cation, it was not feasible to check a configuration in which
all of the parameters were two, let alone three. This was
disappointing.

5 Inserted bugs

To get more assurance that the specification was cor-
rect, we intentionally added bugs to the specification to
see if the model checker could find violations within the
small configurations that were feasible to check. The
idea was to start with the smallest configuration and then
carefully increase the configuration parameters until the
model checker found a violation.

In order to insert a bug, we identified a place in the
specification where it seemed likely that omitting a check
or an action would prove harmful to correct behavior.
Since the intent of a specification is to capture what is nec-
essary for correct behavior, such bugs of omission could
be inserted at almost any point. Table 2 shows the results.
The 16 different bugs we investigated are as follows:

- **BugObtainAccessNoCheckHappy** models what hap-
pens if Pasture fails to bind the key such that it can be
used for decryption only when the secure execution
mode PCR is happy.

- **BugObtainAccessNoCheckSeal** models what hap-
pens if Pasture fails to bind the key such that it can be
used for decryption only when the sealPCR contains
its reboot value.

- **BugProveRevokeNoCheckHappy** models what hap-
pens if Pasture fails to check in a proof of revocation
that the application PCR was quoted at a time when
simultaneously the secure execution mode PCR was
happy.

- **BugProveRevokeNoCheckSeal** models what happens
if Pasture fails to check in a proof of revocation
that the application PCR was quoted at a time when
simultaneously the seal PCR contained its reboot
value.

---

4Assuming all fingerprints are equally likely, the probability of a col-
lision among \( k \) independent probes into a set of size \( H \) can be estimated
as \( 1 - \exp\left(-k \cdot (k - 1)/(2 \cdot H)\right) \). This is known as the birthday
paradox. The formula calculates out to 0.0039 for the given number of
distinct states using 64-bit fingerprints.
### Table 2: Model checking results for inserted bugs. All runs performed on laptop.

<table>
<thead>
<tr>
<th>Bug</th>
<th>config</th>
<th>counterexample (if any) found at</th>
<th>run time</th>
<th>invariant violated</th>
</tr>
</thead>
<tbody>
<tr>
<td>BugObtainAccessNoCheckHappy</td>
<td>1 1 1 1 1</td>
<td>8</td>
<td>1969</td>
<td>4s</td>
</tr>
<tr>
<td>BugObtainAccessNoCheckSeal</td>
<td>1 1 1 1 1</td>
<td>19</td>
<td>29259</td>
<td>7s</td>
</tr>
<tr>
<td>BugProveRevokeNoCheckHappy</td>
<td>1 1 1 1 1</td>
<td>10</td>
<td>5836</td>
<td>3s</td>
</tr>
<tr>
<td>BugProveRevokeNoCheckSeal</td>
<td>1 1 1 1 1</td>
<td>21</td>
<td>37812</td>
<td>7s</td>
</tr>
<tr>
<td>BugRecovNoCheckApp</td>
<td>1 1 1 1 1</td>
<td>19</td>
<td>28013</td>
<td>6s</td>
</tr>
<tr>
<td>BugRecovNoCheckCur</td>
<td>1 1 1 1 1</td>
<td>12</td>
<td>9029</td>
<td>5s</td>
</tr>
<tr>
<td>BugRecovNoClrCur</td>
<td>1 1 1 1 1</td>
<td>12</td>
<td>6368</td>
<td>4s</td>
</tr>
<tr>
<td>BugSealNoExt</td>
<td>1 1 1 1 1</td>
<td>19</td>
<td>109599</td>
<td>15s</td>
</tr>
<tr>
<td>BugChkptNoCheckTsHappy</td>
<td>1 1 1 1 1</td>
<td>20</td>
<td>42021</td>
<td>8s</td>
</tr>
<tr>
<td>BugChkptNoCheckTsSeal</td>
<td>1 1 1 1 1</td>
<td>12</td>
<td>87407</td>
<td>165s</td>
</tr>
<tr>
<td>BugChkptNoCheckTsCtr</td>
<td>1 1 1 1 1</td>
<td>29</td>
<td>107183</td>
<td>16s</td>
</tr>
<tr>
<td>BugChkptSaveCurApp</td>
<td>1 1 1 1 1</td>
<td>20</td>
<td>32826</td>
<td>8s</td>
</tr>
<tr>
<td>BugChkptNoIncCtr</td>
<td>1 1 1 1 1</td>
<td>29</td>
<td>66215</td>
<td>11s</td>
</tr>
<tr>
<td>BugChkptNoSetCur</td>
<td>1 1 1 2 2</td>
<td>32</td>
<td>198270</td>
<td>250s</td>
</tr>
<tr>
<td>BugAuditNoCheckHappy</td>
<td>1 1 1 1 1</td>
<td>9</td>
<td>2490</td>
<td>4s</td>
</tr>
<tr>
<td>BugAuditNoCheckSeal</td>
<td>1 1 2 1 2</td>
<td>34</td>
<td>853554</td>
<td>166s</td>
</tr>
</tbody>
</table>

- **BugRecovNoCheckApp** models what happens if secure execution mode within recovery fails to check that the application PCR was restored to the value saved in the NVRAM.

- **BugRecovNoCheckCur** models what happens if secure execution mode within recovery fails to check that the value saved in the NVRAM is marked as current.

- **BugRecovNoClrCur** models what happens if secure execution mode within recovery fails to clear the current flag in the NVRAM.

- **BugSealNoExt** models what happens if the seal transport session within checkpoint fails to extend the seal PCR.

- **BugChkptNoCheckTsHappy** models what happens if secure execution mode within checkpoint fails to check that the seal attestation recorded that the secure execution mode PCR was happy.

- **BugChkptNoCheckTsSeal** models what happens if secure execution mode within checkpoint fails to check that the seal attestation recorded that the seal PCR contained its reboot value.

- **BugChkptNoCheckTsCtr** models what happens if secure execution mode within checkpoint fails to check that the seal attestation recorded the same value of the boot counter as it currently contains.

- **BugChkptSaveCurApp** models what happens if secure execution mode within checkpoint saves in NVRAM the current application PCR rather than the value of the application PCR recorded in the seal attestation.

- **BugChkptNoIncCtr** models what happens if secure execution mode within checkpoint fails increment the boot counter.

- **BugChkptNoSetCur** models what happens if secure execution mode within checkpoint fails set the current flag in the NVRAM.
5.2 Bugs that were not safety violations

- **BugAuditNoCheckHappy** models what happens if the verifier of an audit fails to check that the application PCR was quoted at a time when simultaneously the secure execution mode PCR was happy.

- **BugAuditNoCheckSeal** models what happens if the verifier of an audit fails to check that the application PCR was quoted at a time when simultaneously the seal PCR contained its reboot value.

5.1 Rapid finding of counterexamples

For all but three bugs the model checker found, within a few seconds of wall clock run time in a very small configuration, a counterexample execution trace that exhibited a violation of an invariant.

For example, consider **BugChkptNoCheckTsCtr**. In this bug, the secure execution mode routine within Checkpoint neglects to check that the SEAL attestation quotes a boot counter value that is the same as the current boot counter value.

The counterexample found by TLC violated the invariant **InvAccessUndeniability** because execution reached a state in which (1) a key was present in the obtains fiduciary variable, meaning that at some point access had been obtained to the key, and (2) the node was auditable but the resulting audit log (based on PCR\textsubscript{APP}) did not include this key. In the TLC counterexample, the state got this way as follows:

1. an initial Recovery sequence,
2. a normal Checkpoint sequence, which performed a SEAL transport session and passed the SEAL attestation to the secure execution mode routine within Checkpoint, which saved the initial, empty log in the NVRAM,
3. a reboot,
4. a normal Recovery sequence,
5. an extension of PCR\textsubscript{APP} to obtain access to a key,
6. an adversarial entry to the secure execution mode routine within Checkpoint, passing it the SEAL attestation from the first Checkpoint sequence, and then performing the routine to save the initial, empty log in the NVRAM as current, (this is where the bug took effect)
7. a reboot, and finally
8. a normal Recovery sequence, which restored PCR\textsubscript{APP} to the value of the empty log, while establishing PCR\textsubscript{SEM} = SemHappy and PCR\textsubscript{SEAL} = SealReboot.

The counterexample requires the boot counter to be incremented twice, which is why the configuration that exhibits the counterexample requires MaxBootCtr = 2.

All of the other counterexamples were found in a minimal configuration that permitted at most one boot counter increment. Since the counterexamples were all found in such very small configurations, it would seem likely that the Pasture node specification does not have any “interesting” behavior that comes out only at higher configuration parameter values. This result gives a reasonable assurance that if there were a bug in the original specification, it would have been found in the original model checking runs.

In prior work [10] we also found that inserted bugs were detected in model checking runs much shorter than the runs required to model check the correct specification with “decent” configuration parameter values, although not nearly to the dramatic extent that we see in the Pasture node specification.

5.2 Bugs that were not safety violations

In three cases the bugs we introduced did not produce counterexamples in small configurations. We examined these bugs more closely and it turned out that they were not safety violations after all.

In the case of **BugChkptNoSetCur**, the checkpoint routine fails to set the current flag in the NVRAM after saving the application PCR. The consequence of this bug is that it will not be possible to recover after a reboot. Although this is a serious liveness problem, it is not a safety violation.

The other two cases, **BugChkptNoCheckTsSeal** and **BugAuditNoCheckSeal** are perhaps more interesting.

In **BugChkptNoCheckTsSeal** the checkpoint SEM routine fails to check that PCR\textsubscript{SEAL} = SealReboot. We can see that this bug results in different execution behavior in the (1,1,2,1,2) configuration because the number of distinct states with the bug in Table 2 is different from the number listed for this configuration in Table 1.

With this bug, the adversary can run the transport ses-
sion to take a SEAL, then perform some additional extensions on PCR_APP, then perform the transport session a second time to take a second SEAL. Since the checkpoint SEM routine fails to check the value of PCR_SEAL in the SEAL, it will accept either of the two SEALs indiscriminately. So with this bug, the adversary can optionally either leave the additional extensions on the log by calling checkpoint with the second SEAL or retract them by calling checkpoint with the first SEAL, in either case afterwards rebooting and recovering in the normal way. This is strange behavior, because Pasture’s design is based on the idea of an append-only log, and this bug permits the adversary to retract some entries from the end of the log.

But although the behavior is strange, it turns out that there is no actual safety violation. The additional extensions performed by the adversary cannot be used to obtain access to any keys or generate any verifiable proofs of revocation, since PCR_SEAL will no longer contain its original SealReboot value after the first transport session runs. The entries that the adversary can retract from the log are merely “phantom” entries that do not correspond to any effective decision.

A similar situation exists in the case of BugAuditNoCheckSeal. This bug permits the adversary to add entries to the end of the log as shown by one audit which a later audit will show as having been retracted. But the retractable entries are “phantom” entries that do not correspond to obtaining access to keys or to generating verifiable proofs of revocation.

Originally, when we placed bugs into the specification, we assumed that they all would lead to safety violations. But in three cases this assumption turned out to be mistaken. One benefit of model checking with known bugs is a better understanding of what actually makes the specification work.

6 Formal proof of correctness

Once we were confident that the Pasture node specification was correct, we proceeded to write a formal correctness proof and check it using the TLA+ Proof System [2].

Appendix C shows the proof. Since the TLA+ Proof System currently cannot handle temporal reasoning, we had to check manually the final step that proves that an invariant always holds. We also omitted numerous tedious proofs about properties of sequences.

The proof is based on the idea that there is always at most one current log and the current log can be domiciled in at most one of three places:

- The current log can be domiciled in PCR_APP, when PCR_SEAL $= \text{SemHappy}$ and PCR_SEAL $= \text{SealReboot}$. This is the situation when the Pasture node is operational and processing decisions to obtain access or revoke access to Pasture decryption keys.
- The current log can be domiciled in a SEAL attestation, when the SEAL quotes PCR_SEAL $= \text{SemHappy}$, PCR_SEAL $= \text{SealReboot}$, and the current boot counter. This is the situation during shutdown after the SEAL transport session has been run but before the SEM checkpoint routine is invoked.
- The current log can be domiciled in the NVRAM, when the current flag is set. This is the situation after shutdown before the node reboots.

The proof wraps this idea up into one master invariant called InvOneLog.

In order to establish InvOneLog, the proof first establishes a number of preliminary invariants showing that all variables contain values of the correct type, that PCR_SEM and PCR_SEAL are managed properly, that SEAL attestations quote a reasonable boot counter value, and that the contents of the fiduciary variables obtains and revokes make sense. Once the master invariant InvOneLog is established, the Pasture safety invariants InvAccessUndeniability and InvVerifiableRevocation follow as corollaries.

Most of the proof is consumed with walking each invariant through all of the action alternatives. Although tedious, writing the proof was straightforward. Counting the time it took to learn how to use the TLA+ Proof System, the proof took two weeks to write. Interestingly, the seL4 microkernel verification project (a far larger effort) also found that invariant reasoning dominated their proof effort [4].

As a side note, the two “phantom” entry non-safety-violation bugs discussed in Section 5.2 each violate the invariant InvOneLog since they permit different versions of the current log to exist at the same time. This shows that the proof is stronger than strictly necessary to establish the correctness of the Pasture node specification.
However, weakening the proof to account for this seems like it would add considerable detail to an already tedious proof.

The Pasture node specification runs 19 pages and the formal proof 68 pages. Memoir also used TLA+ and the TLA+ Proof System and their specification runs 40 pages and formal proof 350 pages [3]. The seL4 project used Haskell and Isabelle and took about 2 person-years to create the specification and 11 person-years to create the proof [4]. So even though a formal specification can be somewhat lengthy, a formal proof of its correctness tends to be much more lengthy.

7 Optimized Pasture

After writing, checking, and proving the correctness of the Pasture specification, we realized that the only function of PCR_SEAL was to indicate that a SEAL transport session had been performed and therefore PCR_APP no longer contained the current log. However, it ought to be possible to use PCR_SEM to implement this indication. Extending PCR_SEM inside the SEAL transport session would destroy SemHappy and have the same effect as using PCR_SEAL.

The only question we had to answer was whether or not the TPM client interface permitted code outside of secure execution mode to extend PCR_SEM. TPMs have locality restrictions that prevent extending certain PCRs at certain times. Fortunately, according to the current TPM PC client interface specification [13], the secure execution mode PCR can be extended in hypervisor or operating system code.

Figure 4 shows the implementation of operations in optimized Pasture. The changes from the initial Pasture implementation are simple: merely delete all references to PCR_SEAL and make the SEAL transport session extend PCR_SEM instead of PCR_SEAL.

Although for simplicity we use the same names—such as SemProtect, SemHappy, BINDKEY, and SEAL—in optimized Pasture as in initial Pasture, the routines are slightly different and therefore any derived cryptographic hash values will be different. We do not think that it would be useful to try to mix Pasture implementations within the same system. The conference paper [5] limits its attention to optimized Pasture.

**Figure 4: Optimized Pasture operations.**

### CreateBoundKey(hM):

\[ R_t \leftarrow TPM_{Read}(PCR_{App}) \]
\[ R_{t+1} \leftarrow SHA1(R_t \ || \ hM) \]
\[
\begin{align*}
K & \leftarrow TPM_{CreateWrapKey}(\{ \\
PCR_{App} = R_{t+1} \ \&\& \\
PCR_{SEM} = \text{SemHappy} \}) \\
\alpha & \leftarrow \text{BINDKEY} \\
KP & \leftarrow (\text{"CreateBoundKey"}, hM, R_t, R_{t+1}, \alpha)
\end{align*}
\]

### ObtainAccess(hM, EM):

append \( hM \) to full log
\[
TPM_{Extend}(PCR_{App}, hM) \\
M \leftarrow TPM_{Unbind}(EM)
\]

### RevokeAccess():

\[ R_t \leftarrow TPM_{Read}(PCR_{App}) \]
append \( \delta \) to full log
\[
TPM_{Extend}(PCR_{App}, \delta) \\
R_{t+1}, S_{t+1}, \alpha \leftarrow TPM_{Quote}(PCR_{App}, PCR_{SEM}) \\
RP \leftarrow (\text{"RevokeAccess"}, \delta, R_t, R_{t+1}, S_{t+1}, \alpha)
\]

### Audit(nonce):

\[
R_t, S_t, \alpha \leftarrow TPM_{Quote}(PCR_{App}, PCR_{SEM}, \text{nonce}) \\
AP \leftarrow (\text{"Audit"}, \text{full log}, R_t, S_t, \text{nonce}, \alpha)
\]

### Recover():

FOR EACH entry \( \Delta \) on full log: \( TPM_{Extend}(PCR_{App}, \Delta) \)
\[
\begin{align*}
\text{If } \text{nv.current} \ \&\& \ \text{nv.R} & = TPM_{Read}(PCR_{App}) \\
\text{THEN} \\
\text{nv.current} & \leftarrow \text{FALSE} \\
TPM_{Extend}(PCR_{SEM}, \text{Happy}) \\
\text{ELSE} \\
TPM_{Extend}(PCR_{SEM}, \text{Unhappy})
\end{align*}
\]

### Checkpoint():

\[ R_t \leftarrow TPM_{Read}(PCR_{App}) \]
\[ S_t \leftarrow TPM_{Read}(PCR_{SEM}) \]
\[ C_t \leftarrow TPM_{ReadCounter}(CTR) \]
\[
\begin{align*}
\alpha & \leftarrow TPM_{Extend}(PCR_{SEM}, \text{Seal}) \\
\text{If } \text{Valid}_{\text{Seal}}(\alpha, R_t, S_t, C_t) \\
& \ \& \ S_t = \text{SemHappy} \\
& \ \& \ C_t = TPM_{ReadCounter}(CTR) \\
\text{THEN} \\
TPM_{IncrementCounter}(CTR) \\
vn.current & \leftarrow \text{TRUE} \\
TPM_{Extend}(PCR_{SEM}, \text{Unhappy})
\end{align*}
\]
Appendix D gives a TLA+ specification of the state within a Pasture node according to optimized Pasture. This specification was created by a simple edit of the initial specification.

To prove the correctness of optimized Pasture, one could prove a refinement mapping showing that every action in optimized Pasture corresponds to an action in initial Pasture. However, since the change in the specification was slight, it was easier just to edit the initial correctness proof.

Appendix E shows the formal proof of correctness for optimized Pasture. Basically, all that was required in editing the initial proof was to delete all references to PCR SEAL and to rework proof steps in three places regarding the changed NextSealTs action. The hierarchical structure of TLA+ proofs was a great benefit here, because the TLA+ Proof System and toolbox made it easy to find the places that had to be changed. Except for temporal reasoning, which the TLA+ Proof System currently cannot handle, and numerous tedious proofs about properties of sequences, all steps in the reworked proof have been mechanically checked.

8 Conclusion

Model checking with inserted bugs provides reasonable confidence that the specification is correct. Examining the counterexample execution traces can lead to improved understanding of the specification and possible improvements. For example, in the Pasture specification, we discovered that the specification could be made weaker, with the incorporation of two “phantom” entry bugs, and still maintain its invariants. However, weakening the specification in this way would make it much more tedious to prove that the invariants were maintained.

Formal proofs give a greater assurance, but they can be tedious. An enormous amount of detail is required to guide a mechanical proof checker through the verification process. When formal proofs of safety are important, it can sometimes be better to adopt a stronger specification than strictly necessary in order to make maintenance of the safety properties easier to prove.

With the TLA+ proof system, the same specification can be both model checked and augmented with a mechanically checked proof. This gives even more confidence that the specification is correct.

Given an existing formal proof of correctness, the TLA+ hierarchical proof structure made it surprisingly easy to create a proof of correctness for a slightly revised specification. A mechanical proof checker gives assurance that the revised proof is correct even though very little intellectual effort had to be expended to create the revised proof.

References


A Specification of Initial Pasture

MODULE PastureNode

EXTENDS Naturals, Sequences, FiniteSets

Override one of the following definitions to introduce a bug in the specification.

\[
\begin{align*}
\text{BugObtainAccessNoCheckHappy} & \triangleq \text{FALSE} \\
\text{BugObtainAccessNoCheckSeal} & \triangleq \text{FALSE} \\
\text{BugProveRevokeNoCheckHappy} & \triangleq \text{FALSE} \\
\text{BugProveRevokeNoCheckSeal} & \triangleq \text{FALSE} \\
\text{BugRecovNoCheckApp} & \triangleq \text{FALSE} \\
\text{BugRecovNoCheckCur} & \triangleq \text{FALSE} \\
\text{BugRecovNoClrCur} & \triangleq \text{FALSE} \\
\text{BugSealNoExt} & \triangleq \text{FALSE} \\
\text{BugChkptNoCheckTsHappy} & \triangleq \text{FALSE} \\
\text{BugChkptNoCheckTsSeal} & \triangleq \text{FALSE} \quad \text{(not actually a safety bug)} \\
\text{BugChkptNoCheckTsCtr} & \triangleq \text{FALSE} \\
\text{BugChkptSaveCurApp} & \triangleq \text{FALSE} \\
\text{BugChkptNoSrcCtr} & \triangleq \text{FALSE} \\
\text{BugChkptNoSetCur} & \triangleq \text{FALSE} \quad \text{liveness bug; not actually a safety bug} \\
\text{BugAuditNoCheckHappy} & \triangleq \text{FALSE} \\
\text{BugAuditNoCheckSeal} & \triangleq \text{FALSE} \quad \text{(not actually a safety bug)}
\end{align*}
\]

PCR INITIALIZATION VALUES

\[
\begin{align*}
\text{CONSTANT PcriAPPBOOT} & \quad \text{reboot initialization of app pcr} \\
\text{CONSTANT PcriSEMBOOT} & \quad \text{reboot initialization of sem pcr} \\
\text{CONSTANT PcriSEMPROTECT} & \quad \text{secure execution mode entry of sem pcr} \\
\text{CONSTANT PcriSEALBOOT} & \quad \text{reboot initialization of seal pcr}
\end{align*}
\]

\[
Pcri \triangleq \\
\{ \\
PcriAPPBOOT, \\
PcriSEMBOOT, \\
PcriSEMPROTECT, \\
PcriSEALBOOT
\}
\]

Initialization of sem pcr via boot and via secure execution mode entry must be different.

\[
\text{ASSUME AssSemProtect} \triangleq \text{PcriSEMBOOT} \neq \text{PcriSEMPROTECT}
\]
PCR EXTENSION VALUES

CONSTANT $PcrxHAPPY$ recover is happy
CONSTANT $PcrxUNHAPPY$ recover is unhappy or checkpoint is unhappy/finished
CONSTANT $PcrxSEAL$ seal marker
CONSTANT $PcrxOBTAIN$ obtain access operation
CONSTANT $PcrxREVOKE$ revoke access operation

$Pcrx \triangleq \{ PcrxHAPPY, PcrxUNHAPPY, PcrxSEAL, PcrxOBTAIN, PcrxREVOKE \}$

Extension for obtain access and extension for revoke access must be different.

ASSUME $AssObtainNeqRevoke \triangleq PcrxOBTAIN \neq PcrxREVOKE$

Extension for happy and extension for unhappy must be different.

ASSUME $AssSemHappy \triangleq PcrxHAPPY \neq PcrxUNHAPPY$

PCR VALUES

A pcr value is modeled as an initialization followed by a sequence of extensions.

$Pcr \triangleq [ \text{init} : Pcri, \text{extq} : \text{Seq}(Pcrx) ]$

Initial pcr value.

$PcrInit(i) \triangleq [ \text{init} \mapsto i, \text{extq} \mapsto \emptyset ]$
Extend a pcr value.

\[
Pcr\text{Extend}(p, x) \triangleq \\
\begin{cases} 
\text{init} \mapsto p.\text{init}, \\
\text{extq} \mapsto \text{Append}(p.\text{extq}, x)
\end{cases}
\]

Number of extensions in a pcr value.

\[
Pcr\text{Len}(p) \triangleq \\
\text{Len}(p.\text{extq})
\]

\(Pcr\ s\ is \leq Pcr\ t.\) This means that with zero or more extensions, you can extend \(s\) to reach \(t.\) This is a partial order relation.

\[
Pcr\text{Leq}(s, t) \triangleq \\
\text{LET} \\
\quad \text{sinit} \triangleq s.\text{init} \\
\quad \text{sextq} \triangleq s.\text{extq} \\
\quad \text{sn} \triangleq \text{Len}(\text{sextq}) \\
\quad \text{tinit} \triangleq t.\text{init} \\
\quad \text{textq} \triangleq t.\text{extq} \\
\quad \text{tn} \triangleq \text{Len}(\text{textq}) \\
\quad \text{uextq} \triangleq \text{SubSeq}(\text{textq}, 1, \text{sn}) \\
\text{IN} \\
\quad \text{sinit} = \text{tinit} \\
\quad \text{sn} \leq \text{tn} \\
\quad \text{sextq} = \text{uextq}
\]

Determine if a pcr value has been extended.

\[
Pcr\text{HasExtension}(p) \triangleq \\
Pcr\text{Len}(p) > 0
\]

Assuming a pcr value has been extended, get the prior pcr value that this one was extended from. We assume the adversary can compute this by watching all pcr computations.

\[
Pcr\text{Prior}(p) \triangleq \\
\text{CASE}\ Pcr\text{HasExtension}(p) \rightarrow \\
\text{LET} \\
\quad n \triangleq \text{Len}(p.\text{extq}) - 1 \\
\text{IN} \\
\quad \text{init} \mapsto p.\text{init}, \\
\quad \text{extq} \mapsto \text{SubSeq}(p.\text{extq}, 1, n)
\]
Assuming a pcr value has been extended, get the last extension. We assume the adversary can compute this by watching all pcr computations.

\[
\text{PcrLastExtension}(p) \triangleq \\
\text{CASE}\ PcrHasExtension(p) \to \\
p.\text{extq}[\text{Len}(p.\text{extq})]
\]

### WELL KNOWN PCR VALUES

**Value of the application pcr attained by rebooting.**

\[
\text{AppReboot} \triangleq \text{PcrInit}(PcriAPPBOOT)
\]

**Value of the secure execution mode pcr attained by rebooting.**

\[
\text{SemReboot} \triangleq \text{PcrInit}(PcriSEMBOOT)
\]

**Value of the secure execution mode pcr attained by entering the protected module in secure execution mode. This value permits access to the Pasture protected \text{Nvram}.**

\[
\text{SemProtect} \triangleq \text{PcrInit}(PcriSEMPROTECT)
\]

**Value of the secure execution mode pcr that indicates that Pasture is happy. Recovery has been properly performed and bound keys may be used. Checkpoint has not yet been invoked.**

\[
\text{SemHappy} \triangleq \text{PcrExtend}(\text{SemProtect}, \text{PcrxHAPPY})
\]

**Value of the seal pcr attained by rebooting.**

\[
\text{SealReboot} \triangleq \text{PcrInit}(PcriSEALBOOT)
\]

### PC VALUES

**anywhere not in secure execution mode**

\[
Pc\text{IDLE} \triangleq \text{"idle"}
\]

**steps in secure execution mode within recover**

\[
Pc\text{RECOV1} \triangleq \text{"recov1"}
Pc\text{RECOV2} \triangleq \text{"recov2"}
Pc\text{RECOV3} \triangleq \text{"recov3"}
Pc\text{Recov} \triangleq \{Pc\text{RECOV1}, Pc\text{RECOV2}, Pc\text{RECOV3}\}
\]

**steps in secure execution mode within checkpoint**

\[
Pc\text{CHKPT1} \triangleq \text{"chkpt1"}
\]
\( \text{PcCHKPT2} \triangleq \text{"chkpt2"} \)
\( \text{PcCHKPT3} \triangleq \text{"chkpt3"} \)
\( \text{PcCHKPT4} \triangleq \text{"chkpt4"} \)
\( \text{PcCHKPT5} \triangleq \text{"chkpt5"} \)
\( \text{PcChkpt} \triangleq \{ \text{PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5} \} \)

\( \text{Pc} \triangleq \{ \text{PcIDLE} \} \cup \text{PcRecov} \cup \text{PcChkpt} \)

PROTECTED NV RAM STATE

\( \text{Nv} \triangleq \)
\[
\begin{array}{l}
\{ \text{appPcr : Pcr, } \text{copy of the application pcr} \\
\text{current : BOOLEAN } \text{copy of application pcr is current} \\
\}
\end{array}
\]

\( \text{InitNv} \triangleq \)
\[
\begin{array}{l}
\{ \text{appPcr }\mapsto\text{AppReboot, } \\
\text{current }\mapsto\text{TRUE} \\
\}
\end{array}
\]

SEAL OPERATION TRANSPORT SESSION STATE

We model the signed "seal operation" transport session as a record of the input values required in order for the transport session TPM signature to be valid.

\( \text{SignedTs} \triangleq \)
\[
\begin{array}{l}
\{ \text{semPcr : Pcr, } \text{copy of the secure execution mode pcr on entry} \\
\text{sealPcr : Pcr, } \text{copy of the seal pcr on entry} \\
\text{appPcr : Pcr, } \text{copy of the application pcr on entry} \\
\text{bootCtr : Nat } \text{copy of the reboot counter on entry} \\
\}
\end{array}
\]

The adversary cannot forge a correctly signed seal attestation. We model all incorrectly signed ones as the following single value.

\( \text{NullTs} \triangleq \text{CHOOSE NullTs : NullTs } \notin \text{SignedTs} \)
\[ Ts \triangleq SignedTs \cup \{NullTs\} \]

**STATE**

<table>
<thead>
<tr>
<th>VARIABLE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( nv )</td>
<td>Pasture’s protected NV RAM region</td>
</tr>
<tr>
<td>( appPcr )</td>
<td>the application pcr</td>
</tr>
<tr>
<td>( semPcr )</td>
<td>the secure execution mode pcr</td>
</tr>
<tr>
<td>( sealPcr )</td>
<td>the seal pcr</td>
</tr>
<tr>
<td>( bootCtr )</td>
<td>the reboot counter</td>
</tr>
<tr>
<td>( pc )</td>
<td>the pc</td>
</tr>
<tr>
<td>( chkptts )</td>
<td>ts passed to sem within checkpoint</td>
</tr>
<tr>
<td>( tsvalues )</td>
<td>what ts values are known</td>
</tr>
<tr>
<td>( obtains )</td>
<td>decisions to obtain access</td>
</tr>
<tr>
<td>( revokes )</td>
<td>decisions to prove revoke access</td>
</tr>
</tbody>
</table>

Tuple of all variables.

\[ vars \triangleq (nv, appPcr, semPcr, sealPcr, bootCtr, pc, chkptts, tsvalues, obtains, revokes) \]

**STATE PREDICATES**

The node is currently in secure execution mode.

\[ InSem \triangleq pc \neq PcIDLE \]

**NEXT STATE RELATION**

Employ a key binding to obtain access to read a message.
If the last extension to the application pcr was an OBTAIN operation, then in full generality there could have been a key bound to this application pcr value. So record the information that we obtained access to this key binding.

\[
\text{NextObtainAccess} \triangleq
\]

\[
\text{LET}
\]

\[
\begin{align*}
\text{pcr}_1 & \triangleq \text{appPcr} & \text{current app pcr} \\
\text{pcr}_0 & \triangleq \text{PcrPrior}(\text{pcr}_1) & \text{prior app pcr} \\
x & \triangleq \text{PcrLastExtension}(\text{pcr}_1) & \text{presumed OBTAIN extension}
\end{align*}
\]

\[
\text{IN}
\]

\[
\begin{align*}
\land \neg \text{InSem} & \quad \text{must not be in secure execution mode} \\
\land \text{PcrHasExtension}(\text{pcr}_1) & \quad \text{have an extension} \\
x & = \text{PcrOBTAIN} & \text{last extension was OBTAIN}
\end{align*}
\]

It is a bug to fail to bind the key such that it can be used for decryption only when the secure execution mode pcr is happy.

\[
\begin{align*}
\land & \text{IF BugObtainAccessNoCheckHappy THEN TRUE ELSE} \\
\text{semPcr} & = \text{SemHappy}
\end{align*}
\]

It is a bug to fail to bind the key such that it can be used for decryption only when the seal pcr is in the reboot value.

\[
\begin{align*}
\land & \text{IF BugObtainAccessNoCheckSeal THEN TRUE ELSE} \\
\text{sealPcr} & = \text{SealReboot}
\end{align*}
\]

\[
\land \text{obtains' = obtains } \cup \{\text{pcr}_1\}
\land \text{UNCHANGED } w
\land \text{UNCHANGED appPcr}
\land \text{UNCHANGED semPcr}
\land \text{UNCHANGED sealPcr}
\land \text{UNCHANGED bootCtr}
\land \text{UNCHANGED pc}
\land \text{UNCHANGED chkptts}
\land \text{UNCHANGED tsvalues}
\land \text{UNCHANGED revokes}
\]

Construct a proof of revocation.

If the last extension to the application pcr was a REVOKE operation, then in full generality there could have been a key bound to the pcr value in which instead the last extension was an OBTAIN. But by extending with a REVOKE we have instead revoked the key binding. So record the information that we could construct a proof of revocation.

A proof of revocation consists of the following exhibits:

(a) \(\text{pcr}_0\), a purported prior application \(\text{pcr}\) value
(b) \(\text{pcr}_1r\), a purported current application \(\text{pcr}\) value
(c) \(x\), the REVOKE extension satisfying \(\text{pcr}_1r = \text{PcrExtend}(\text{pcr}_0, x)\)
(d) a quote of the application \(\text{pcr} = \text{pcr}_1r\) with the sem \(\text{pcr} = \text{SemHappy}\) and the seal \(\text{pcr} = \text{SealReboot}\).

These exhibits suffice to prove revocation of any valid key binding to the application \(\text{pcr}\) value \(\text{PcrExtend}(\text{pcr}_0, \text{OBTAIN})\).

\[
\text{NextProveRevoke} \triangleq
\]

\[
\text{LET}
\]

\[
\begin{align*}
\text{pcr}_1r & \triangleq \text{appPcr} & \text{current app pcr}
\end{align*}
\]
\[ pcr0 \equiv \text{PcrPrior}(pcr1r) \quad \text{prior app pcr} \]
\[ x \equiv \text{PcrLastExtension}(pcr1r) \quad \text{presumed REVOKE extension} \]

IN
\[ \neg \text{InSem} \quad \text{must not be in secure execution mode} \]
\[ \text{PcrHasExtension}(pcr1r) \quad \text{have an extension} \]
\[ x = \text{PcrREVOKE} \quad \text{last extension was a REVOKE} \]

It is a bug to fail to require the proof of revocation to quote the fact that the secure execution mode pcr is happy.
\[ \land \text{IF BugProveRevokeNoCheckHappy THEN TRUE ELSE} \]
\[ \text{semPcr} = \text{SemHappy} \]

It is a bug to fail to require the proof of revocation to quote the fact that the seal pcr is in the reboot value.
\[ \land \text{IF BugProveRevokeNoCheckSeal THEN TRUE ELSE} \]
\[ \text{sealPcr} = \text{SealReboot} \]
\[ \land \text{revokes}' = \text{revokes} \cup \{pcr1r\} \]
\[ \land \text{UNCHANGED nv} \]
\[ \land \text{UNCHANGED appPcr} \]
\[ \land \text{UNCHANGED semPcr} \]
\[ \land \text{UNCHANGED sealPcr} \]
\[ \land \text{UNCHANGED bootCtr} \]
\[ \land \text{UNCHANGED pc} \]
\[ \land \text{UNCHANGED chkptts} \]
\[ \land \text{UNCHANGED tsvalues} \]
\[ \land \text{UNCHANGED obtains} \]

Reboot the node.

This can happen at absolutely any time, due to adversarial action. However, if it happens without going through the proper seal and checkpoint actions, liveness may be lost.

Resetting \text{chkptts} to its initial value erases information and thus reduces the number of distinct states that model checking has to explore. But note that the adversary could always remember whatever value chkptts had before and call sem checkpoint with that value.

NextReboot \[ \equiv \]
\[ \land \text{appPcr}' = \text{AppReboot} \]
\[ \land \text{semPcr}' = \text{SemReboot} \]
\[ \land \text{sealPcr}' = \text{SealReboot} \]
\[ \land \text{pc}' = \text{PcIDLE} \]
\[ \land \text{chkptts}' = \text{NullTs} \]
\[ \land \text{UNCHANGED nv} \]
\[ \land \text{UNCHANGED bootCtr} \]
\[ \land \text{UNCHANGED tsvalues} \]
\[ \land \text{UNCHANGED obtains} \]
\[ \land \text{UNCHANGED revokes} \]
Forget one of the seal transport sessions. This can happen at absolutely any time, and represents a loss of knowledge by the adversary which enables additional execution paths to fall within the model checking constraints.

\[
\text{NextForgetSealTs} \triangleq \\
\exists ts \in tsvalues : \\
\land tsvalues' = tsvalues \setminus \{ts\} \\
\land \text{UNCHANGED } nv \\
\land \text{UNCHANGED } appPcr \\
\land \text{UNCHANGED } semPcr \\
\land \text{UNCHANGED } sealPcr \\
\land \text{UNCHANGED } bootCtr \\
\land \text{UNCHANGED } pc \\
\land \text{UNCHANGED } chkptts \\
\land \text{UNCHANGED } obtains \\
\land \text{UNCHANGED } revokes
\]

Extend application pcr arbitrarily.

In proper execution, this action is performed as necessary after reboot to re-extend the application pcr to its last checkpoint value.

In proper execution, this action is performed as desired to decide upon reading or deleting messages.

The adversary can perform this action at any idle time.

\[
\text{NextExtendAppPcr} \triangleq \\
\land \neg \text{InSem } \text{ must not be in secure execution mode} \\
\land \exists x \in Pcrx : \\
\land appPcr' = PcrExtend(appPcr, x) \\
\land \text{UNCHANGED } nv \\
\land \text{UNCHANGED } semPcr \\
\land \text{UNCHANGED } sealPcr \\
\land \text{UNCHANGED } bootCtr \\
\land \text{UNCHANGED } pc \\
\land \text{UNCHANGED } chkptts \\
\land \text{UNCHANGED } tsvalues \\
\land \text{UNCHANGED } obtains \\
\land \text{UNCHANGED } revokes
\]

Extend secure execution mode pcr arbitrarily, due to adversarial action.

\[
\text{NextExtendSemPcr} \triangleq \\
\land \neg \text{InSem } \text{ must not be in secure execution mode} \\
\land \exists x \in Pcrx : \\
\land semPcr' = PcrExtend(semPcr, x) \\
\land \text{UNCHANGED } nv
\]
\(\wedge\) UNCHANGED \(appPcr\)
\(\wedge\) UNCHANGED \(sealPcr\)
\(\wedge\) UNCHANGED \(bootCtr\)
\(\wedge\) UNCHANGED \(pc\)
\(\wedge\) UNCHANGED \(chkptts\)
\(\wedge\) UNCHANGED \(tsvalues\)
\(\wedge\) UNCHANGED \(obtains\)
\(\wedge\) UNCHANGED \(revokes\)

**Extend seal pcr arbitrarily, due to adversarial action.**

\(Next\) \(\text{ExtendSealPcr} \triangleq\)
\(\wedge \neg InSem\)
\(\wedge \exists x \in Pcrx :\)
\(\wedge sealPcr' = PcrExtend(sealPcr, x)\)
\(\wedge\) UNCHANGED \(nv\)
\(\wedge\) UNCHANGED \(appPcr\)
\(\wedge\) UNCHANGED \(semPcr\)
\(\wedge\) UNCHANGED \(bootCtr\)
\(\wedge\) UNCHANGED \(pc\)
\(\wedge\) UNCHANGED \(chkptts\)
\(\wedge\) UNCHANGED \(tsvalues\)
\(\wedge\) UNCHANGED \(obtains\)
\(\wedge\) UNCHANGED \(revokes\)

**Increment reboot counter arbitrarily, due to adversarial action.**

\(Next\) \(\text{IncBootCtr} \triangleq\)
\(\wedge \neg InSem\)
\(\wedge bootCtr' = bootCtr + 1\)
\(\wedge\) UNCHANGED \(nv\)
\(\wedge\) UNCHANGED \(appPcr\)
\(\wedge\) UNCHANGED \(semPcr\)
\(\wedge\) UNCHANGED \(sealPcr\)
\(\wedge\) UNCHANGED \(pc\)
\(\wedge\) UNCHANGED \(chkptts\)
\(\wedge\) UNCHANGED \(tsvalues\)
\(\wedge\) UNCHANGED \(obtains\)
\(\wedge\) UNCHANGED \(revokes\)
Enter secure execution mode within recovery.

In proper execution, this action is performed during system boot after the application PCR has been re-extended to its last checkpoint value. This re-extension is performed by untrusted code that reads the necessary extension values from a stable log.

The adversary can perform this action at any idle time. But it will not do any good unless the application PCR contains the last checkpoint value and the last checkpoint value is marked as current.

\[ NextEnterSemRecov \triangleq \]
\[ \\neg InSem \]
\[ \land semPcr' = SemProtect \]
\[ \land pc' = PcRECOV1 \]
\[ \land UNCHANGED nv \]
\[ \land UNCHANGED appPcr \]
\[ \land UNCHANGED sealPcr \]
\[ \land UNCHANGED bootCtr \]
\[ \land UNCHANGED chkptts \]
\[ \land UNCHANGED tsvalues \]
\[ \land UNCHANGED obtains \]
\[ \land UNCHANGED revokes \]

Predicate for correct entry to secure execution mode within recovery.

\[ EnterSemRecovPredicate \triangleq \]
\[ \land IF BugRecovNoCheckApp THEN TRUE ELSE \]
\[ \land IF BugRecovNoCheckCur THEN TRUE ELSE \]
\[ \land \quad nv.appPcr = appPcr \]
\[ \land \quad nv.current \]

Secure execution mode within recovery step 1, when there is correct entry.

\[ NextSemRecov1WhenCorrect \triangleq \]
\[ \land pc = PcRECOV1 \]
\[ \land EnterSemRecovPredicate \]
\[ \land pc' = PcRECOV2 \]
\[ \land UNCHANGED nv \]
\[ \land UNCHANGED appPcr \]
\[ \land UNCHANGED semPcr \]
\[ \land UNCHANGED sealPcr \]
\[ \land UNCHANGED bootCtr \]
\[ \land UNCHANGED chkptts \]
\[ \land UNCHANGED tsvalues \]
\[ \land UNCHANGED obtains \]
\[ \land UNCHANGED revokes \]
Secure execution mode within recovery step 1, when there is incorrect entry.

\[
\text{NextSemRecov1 WhenIncorrect } \triangleq \n\begin{align*}
&\land pc = PcRECOV1 \\
&\land \neg \text{EnterSemRecovPredicate} \\
&\land \text{semPcr'} = \text{PcrExtend(semPcr, PcrxUNHAPPY)} \\
&\land pc' = PcIDLE \\
&\land \text{UNCHANGED } nv \\
&\land \text{UNCHANGED appPcr} \\
&\land \text{UNCHANGED sealPcr} \\
&\land \text{UNCHANGED bootCtr} \\
&\land \text{UNCHANGED chkptts} \\
&\land \text{UNCHANGED tsvalues} \\
&\land \text{UNCHANGED obtains} \\
&\land \text{UNCHANGED revokes}
\end{align*}
\]

Secure execution mode within recovery step 2. Record that the \( nv \) app pcr might no longer be current.

\[
\text{NextSemRecov2 } \triangleq \\
\text{LET } \\
\quad \text{nvcurent1 } \triangleq \\
\qquad \text{It is a bug for recovery to fail to clear the } nv \text{ ram current flag.} \\
\qquad \text{IF } \text{BugRecovNoClrCur THEN } nv\text{.current ELSE FALSE IN} \\
\qquad \land pc = PcRECOV2 \\
\qquad \land nv' = [nv \text{ EXCEPT } !.\text{current} = nvcurent1] \\
\qquad \land pc' = PcRECOV3 \\
\qquad \land \text{UNCHANGED appPcr} \\
\qquad \land \text{UNCHANGED semPcr} \\
\qquad \land \text{UNCHANGED sealPcr} \\
\qquad \land \text{UNCHANGED bootCtr} \\
\qquad \land \text{UNCHANGED chkptts} \\
\qquad \land \text{UNCHANGED tsvalues} \\
\qquad \land \text{UNCHANGED obtains} \\
\qquad \land \text{UNCHANGED revokes}
\]

Secure execution mode within recovery step 3. Declare correct recovery happiness and exit secure execution mode.

\[
\text{NextSemRecov3 } \triangleq \\
\land pc = PcRECOV3 \\
\land \text{semPcr'} = \text{PcrExtend(semPcr, PcrxHAPPY)} \\
\land pc' = PcIDLE \\
\land \text{UNCHANGED } nv \\
\land \text{UNCHANGED appPcr} \\
\land \text{UNCHANGED sealPcr}
\]
A SPECIFICATION OF INITIAL PASTURE

\[ \begin{align*}
&\bigwedge \text{UNCHANGED } \text{bootCtr} \\
&\bigwedge \text{UNCHANGED } \text{chkptts} \\
&\bigwedge \text{UNCHANGED } \text{tsvalues} \\
&\bigwedge \text{UNCHANGED } \text{obtains} \\
&\bigwedge \text{UNCHANGED } \text{revokes}
\end{align*} \]

Perform a “seal operation” and remember the signed transport session.

In proper execution, provided that the secure execution mode pcr shows that recovery was happy, this action is performed as part of checkpoint during system shutdown. Secure execution mode within checkpoint is then invoked with this transport session as data.

This transport session reads the values of the secure execution mode pcr, the application pcr, the seal pcr, and the reboot counter. Then the seal pcr is extended so that no key bindings will be available until the next happy recovery.

The adversary can record all of the signed transport sessions and try to replay an earlier one to convince secure execution mode within checkpoint to save an old application pcr as “current”. Reading the reboot counter here, and incrementing it in secure execution mode within checkpoint, prevents that.

The adversary might try to advance the application pcr so as to read a message or produce a proof of deletion after the “seal operation” and then invoke secure execution mode within checkpoint and then reboot to roll back the application pcr. Extending the seal pcr prevents that.

NextSealTs \( \triangleq \)

\[
\begin{align*}
\text{LET} \\
\text{ts} \triangleq \\
\left[ \\
\begin{array}{ll}
\text{semPcr} \mapsto \text{semPcr}, & \text{sem pcr on entry} \\
\text{appPcr} \mapsto \text{appPcr}, & \text{app pcr on entry} \\
\text{sealPcr} \mapsto \text{sealPcr}, & \text{seal pcr on entry} \\
\text{bootCtr} \mapsto \text{bootCtr} & \text{reboot ctr on entry}
\end{array}
\right]
\end{align*}
\]

\[
\text{sealPcr} \triangleq \text{sealPcr}^1 \triangleq
\begin{align*}
\text{IF } \text{BugSealNoExt} \text{ THEN } \text{sealPcr} \text{ ELSE } \\
\text{PcrExtend}(\text{sealPcr}, \text{PcrxSEAL})
\end{align*}
\]

IN

\[
\begin{align*}
&\bigwedge \neg \text{InSem} \quad \text{must not be in secure execution mode} \\
&\bigwedge \text{tsvalues}' = \text{tsvalues} \cup \{\text{ts}\} \\
&\bigwedge \text{sealPcr}' = \text{sealPcr}^1 \\
&\bigwedge \text{UNCHANGED } \text{nv} \\
&\bigwedge \text{UNCHANGED } \text{appPcr} \\
&\bigwedge \text{UNCHANGED } \text{semPcr} \\
&\bigwedge \text{UNCHANGED } \text{bootCtr} \\
&\bigwedge \text{UNCHANGED } \text{pc} \\
&\bigwedge \text{UNCHANGED } \text{chkptts} \\
&\bigwedge \text{UNCHANGED } \text{obtains} \\
&\bigwedge \text{UNCHANGED } \text{revokes}
\end{align*}
\]
Enter secure execution mode within checkpoint.

In proper execution, this action is performed during system shutdown following the seal transport session action.

The adversary can perform this action at any idle time, feeding it any known seal transport session value.

\[ \text{NextEntrySemChkpt} \triangleq \]
\[ \land \neg \text{InSem} \quad \text{must not be in secure execution mode} \]
\[ \land \exists ts \in tsvalues : \quad \text{any known ts value} \]
\[ \land \text{semPcr}' = \text{SemProtect} \]
\[ \land pc' = \text{PcCHKPT1} \]
\[ \land \text{chkptts}' = ts \]
\[ \land \text{UNCHANGED} \ nv \]
\[ \land \text{UNCHANGED} \ appPcr \]
\[ \land \text{UNCHANGED} \ sealPcr \]
\[ \land \text{UNCHANGED} \ tsvalues \]
\[ \land \text{UNCHANGED} \ obtains \]
\[ \land \text{UNCHANGED} \ 

Predicate for correct entry to secure execution mode within checkpoint.

\[ \text{EnterSemChkptPredicate} \triangleq \]
\[ \land \text{chkptts} \in \text{SignedTs} \]
\[ \land \text{IF} \ BugChkptNoCheckTsHappy \text{ THEN TRUE ELSE} \]
\[ \text{chkptts}, \text{semPcr} = \text{SemHappy} \]
\[ \land \text{IF} \ BugChkptNoCheckTsSeal \text{ THEN TRUE ELSE} \]
\[ \text{chkptts}, \text{sealPcr} = \text{SealReboot} \]
\[ \land \text{IF} \ BugChkptNoCheckTsCtr \text{ THEN TRUE ELSE} \]
\[ \text{chkptts}, \text{bootCtr} = \text{bootCtr} \]

Secure execution mode within checkpoint step 1, when there is correct entry.

\[ \text{NextSemChkp1 WhenCorrect} \triangleq \]
\[ \land pc = \text{PcCHKPT1} \]
\[ \land \text{EnterSemChkptPredicate} \]
\[ \land pc' = \text{PcCHKPT2} \]
\[ \land \text{UNCHANGED} \ nv \]
\[ \land \text{UNCHANGED} \ appPcr \]
\[ \land \text{UNCHANGED} \ semPcr \]
\[ \land \text{UNCHANGED} \ sealPcr \]
\[ \land \text{UNCHANGED} \ bootCtr \]
\[ \land \text{UNCHANGED} \ 

It is a bug to fail to check that the seal operation recorded that the secure execution mode pcr was happy.

It is a bug to fail to check that the seal operation recorded that the seal pcr was in the reboot value.

It is a bug to fail to check that the seal operation recorded a reboot counter value that matches the current reboot counter.
\(\land\) UNCHANGED \(tsvalues\)
\(\land\) UNCHANGED \(obtains\)
\(\land\) UNCHANGED \(revokes\)

Secure execution mode within checkpoint step 1, when there is incorrect entry.

NextSemChkpt1 WhenIncorrect \(\triangleq\)
\(\land\ pc = Pc\CHKPT1\)
\(\land\ ¬EnterSemChkptPredicate\)
\(\land\ semPcr' = PerExtend(semPcr, PcrxUNHAPPY)\)
\(\land\ pc' = Pc\IDLE\)
\(\land\) UNCHANGED \(nv\)
\(\land\) UNCHANGED \(appPcr\)
\(\land\) UNCHANGED \(sealPcr\)
\(\land\) UNCHANGED \(bootCtr\)
\(\land\) UNCHANGED \(chkptts\)
\(\land\) UNCHANGED \(tsvalues\)
\(\land\) UNCHANGED \(obtains\)
\(\land\) UNCHANGED \(revokes\)

Secure execution mode within checkpoint step 2. Save in \(nv\) appPtr the app ptr recorded at \(ts\) entry.

NextSemChkpt2 \(\triangleq\)
LET
\(nvappPcr1 \triangleq\)
It is a bug for secure execution mode within checkpoint to save in the \(nv\) ram the current application pcr rather than the seal operation’s recorded application pcr.
IF BugChkptSaveCurApp THEN appPcr ELSE chkptts.appPcr
IN
\(\land\ pc = Pc\CHKPT2\)
\(\land\ nv' = [nv\ EXCEPT !.appPcr = nvappPcr1]\)
\(\land\ pc' = Pc\CHKPT3\)
\(\land\) UNCHANGED \(appPcr\)
\(\land\) UNCHANGED \(semPcr\)
\(\land\) UNCHANGED \(sealPcr\)
\(\land\) UNCHANGED \(bootCtr\)
\(\land\) UNCHANGED \(chkptts\)
\(\land\) UNCHANGED \(tsvalues\)
\(\land\) UNCHANGED \(obtains\)
\(\land\) UNCHANGED \(revokes\)

Secure execution mode within checkpoint step 3. Prevent a \(ts\) replay by incrementing the reboot ctr.
NextSemChkpt3 =

LET

\[ \text{let } \begin{align*}
\text{bootCtr}_1 &= \text{let } \\
\end{align*} \]

It is a bug for secure execution mode within checkpoint to fail to increment the reboot counter.

IF BugChkptNoIncCtr THEN bootCtr ELSE bootCtr + 1

IN

\[ \land pc = \text{PcCHKPT3} \]
\[ \land \text{bootCtr}' = \text{bootCtr}_1 \]
\[ \land pc' = \text{PcCHKPT4} \]
\[ \land \text{UNCHANGED } \text{nv} \]
\[ \land \text{UNCHANGED } \text{appPcr} \]
\[ \land \text{UNCHANGED } \text{semPcr} \]
\[ \land \text{UNCHANGED } \text{sealPcr} \]
\[ \land \text{UNCHANGED } \text{chkptts} \]
\[ \land \text{UNCHANGED } \text{tsvalues} \]
\[ \land \text{UNCHANGED } \text{obtains} \]
\[ \land \text{UNCHANGED } \text{revokes} \]

Secure execution mode within checkpoint step 4. Declare that the \text{nv appPcr} is current so that after reboot recovery will be able to succeed.

NextSemChkpt4 =

LET

\[ \text{nvcurrent}_1 = \text{let } \]

It is a bug for secure execution mode within checkpoint to fail to set the \text{NV RAM current} flag.

Actually, this bug does not result in a safety violation.

IF BugChkptNoSetCur THEN \text{nv.current} ELSE TRUE

IN

\[ \land pc = \text{PcCHKPT4} \]
\[ \land \text{nv}' = [\text{nv EXCEPT } !.\text{current} = \text{nvcurrent}_1] \]
\[ \land pc' = \text{PcCHKPT5} \]
\[ \land \text{UNCHANGED } \text{appPcr} \]
\[ \land \text{UNCHANGED } \text{semPcr} \]
\[ \land \text{UNCHANGED } \text{sealPcr} \]
\[ \land \text{UNCHANGED } \text{bootCtr} \]
\[ \land \text{UNCHANGED } \text{chkptts} \]
\[ \land \text{UNCHANGED } \text{tsvalues} \]
\[ \land \text{UNCHANGED } \text{obtains} \]
\[ \land \text{UNCHANGED } \text{revokes} \]

Secure execution mode within checkpoint step 5. Extend \text{sem pcr} with unhappy so protected \text{nv ram} will be inaccessible.

NextSemChkpt5 =

\[ \land pc = \text{PcCHKPT5} \]
A SPECIFICATION OF INITIAL PASTURE

\[ \land semPcr' = PcrExtend(semPcr, PcrxUNHAPPY) \]
\[ \land pc' = PcIDLE \]
\[ \land UNCHANGED nv \]
\[ \land UNCHANGED appPcr \]
\[ \land UNCHANGED sealPcr \]
\[ \land UNCHANGED bootCtr \]
\[ \land UNCHANGED chkptts \]
\[ \land UNCHANGED tsvalues \]
\[ \land UNCHANGED obtains \]
\[ \land UNCHANGED revokes \]

SPECIFICATION

\( \text{Init} \triangleq \)
\[ \land nv = InitNv \]
\[ \land appPcr = AppReboot \]
\[ \land semPcr = SemReboot \]
\[ \land sealPcr = SealReboot \]
\[ \land bootCtr = 0 \]
\[ \land pc = PcIDLE \]
\[ \land chkptts = NullTs \]
\[ \land tsvalues = \{ NullTs \} \]
\[ \land obtains = \{ \} \]
\[ \land revokes = \{ \} \]

\( \text{Next} \triangleq \)
\[ \lor NextObtainAccess \]
\[ \lor NextProveRevoke \]
\[ \lor NextReboot \]
\[ \lor NextForgetSealTs \]
\[ \lor NextExtendAppPcr \]
\[ \lor NextExtendSemPcr \]
\[ \lor NextExtendSealPcr \]
\[ \lor NextIncBootCtr \]
\[ \lor NextEnterSemRecov \]
\[ \lor NextSemRecov1 WhenCorrect \]
\[ \lor NextSemRecov1 WhenIncorrect \]
\[ \lor NextSemRecov2 \]
\[ \lor NextSemRecov3 \]

anybody can create a NullTs
\( \lor \text{NextSealTs} \)
\( \lor \text{NextEnterSemChkpt} \)
\( \lor \text{NextSemChkpt1 WhenCorrect} \)
\( \lor \text{NextSemChkpt1 WhenIncorrect} \)
\( \lor \text{NextSemChkpt2} \)
\( \lor \text{NextSemChkpt3} \)
\( \lor \text{NextSemChkpt4} \)
\( \lor \text{NextSemChkpt5} \)

\[ \text{Spec} \triangleq \text{Init} \land \Box [\text{Next}]_{\text{vars}} \]

**INVARIANTS**

**Type invariant.**

\[ \text{InvType} \triangleq \]
\[ \land \text{nv:: } \text{nv} \in \text{Nv} \]
\[ \land \text{appPcr:: } \text{appPcr} \in \text{Pcr} \]
\[ \land \text{semPcr:: } \text{semPcr} \in \text{Pcr} \]
\[ \land \text{sealPcr:: } \text{sealPcr} \in \text{Pcr} \]
\[ \land \text{bootCtr:: } \text{bootCtr} \in \text{Nat} \]
\[ \land \text{pc:: } \text{pc} \in \text{Pc} \]
\[ \land \text{chkptts:: } \land \text{chkptts} \in \text{Ts} \]
\[ \land \text{pc} \in \text{PcChkpt} \setminus \{\text{PcCHKPT1}\} \Rightarrow \text{chkptts} \in \text{SignedTs} \]
\[ \land \text{tvalues:: } \text{tvalues} \in \text{SUBSET} \text{ Ts} \]
\[ \land \text{obtains:: } \text{obtains} \in \text{SUBSET} \text{ Pcr} \]
\[ \land \text{revokes:: } \text{revokes} \in \text{SUBSET} \text{ Pcr} \]

\( Nv \) protection invariant.

Being in secure execution mode is equivalent to saying that the secure execution mode pcr permits access to protected \( Nv \) ram.

\[ \text{InvNuProtection} \triangleq \]
\[ \text{InSem} \equiv (\text{semPcr} = \text{SemProtect}) \]

Verifiable revocation invariant. There had better not be any decisions to obtain access for which a proof of revocation was also constructed.

\[ \text{InvVerifiableRevocation} \triangleq \]
\[ \forall o \in \text{obtains} : \text{last extension was OBTAIN} \]
\[ \forall r \in \text{revokes} : \text{last extension was REVOKE} \]
\[ \text{PcrPrior}(o) \neq \text{PcrPrior}(r) \quad \text{cannot have both extended from same place} \]

Access undeniability.

This invariant is modeled as performing an audit on the present state and seeing that all key bindings that have been used to obtain access appear in the audit report. A key binding \( o \) appears in the audit report iff \( \text{PcrLeq}(o, \text{appPcr}) \), which means that there exists a sequence of zero or more extensions from \( o \) that reach \( \text{appPcr} \).

However, it might be impossible to generate a valid audit report in the present node state. That is okay.

\[ \text{InvAccessUndeniability} \triangleq \]

\[ \left( \text{It is a bug to fail to require the audit to quote } \text{SemHappy} \right) \]
\[ \land \quad \left( \text{If } \text{BugAuditNoCheckHappy} \text{ THEN TRUE ELSE } \text{semPcr} = \text{SemHappy} \right) \]
\[ \left( \text{It is a bug to fail to require the audit to quote } \text{SealReboot} \right) \]
\[ \land \quad \left( \text{If } \text{BugAuditNoCheckSeal} \text{ THEN TRUE ELSE } \text{sealPcr} = \text{SealReboot} \right) \]
\[ \Rightarrow \]
\[ \forall o \in \text{obtains} : \text{PcrLeq}(o, \text{appPcr}) \]
B Model

MODULE PastureNodeModel

VARIABLE nv  Pasture's protected NV RAM region
VARIABLE appPcr  the application pcr
VARIABLE semPcr  the secure execution mode pcr
VARIABLE sealPcr  the seal pcr
VARIABLE bootCtr  the reboot counter
VARIABLE pc  pc
VARIABLE chkpts  ts passed to sem within checkpoint
VARIABLE tsvalues  what ts values are known
VARIABLE obtains  decisions to obtain access
VARIABLE revokes  decisions to prove revoke access

INSTANCE PastureNode
WITH
  PcriAPPBOOT ← "boot",
  SEMBOOT and SEMPROTECT must be different.
  PcriSEMBOOT ← "boot",
  PcriSEMPROTECT ← "protect",
  PcriSEALBOOT ← "boot",
  HAPPY and UNHAPPY must be different.
  PcrxHAPPY ← 0,
  PcrxUNHAPPY ← 1,
  OBTAIN and REVOKE must be different.
  PcrxOBTAIN ← 0,
  PcrxREVOKE ← 1,
  PcrxSEAL ← 0

MODEL-CHECKING CONSTRAINT

Override these definitions to adjust the constraint.
MaxAppPcrLen ≜ 1
MaxSemPcrLen ≜ 1
MaxSealPcrLen ≜ 1
MaxTsValues ≜ 1
MaxBootCtr ≜ 1
\[
\text{Constrain} \overset{\Delta}{=} \\
\land \ PcrLen(appPcr) \leq \text{MaxAppPcrLen} \\
\land \ PcrLen(semPcr) \leq \text{MaxSemPcrLen} \\
\land \ PcrLen(sealPcr) \leq \text{MaxSealPcrLen} \\
\land \ \text{Cardinality}(tsvalues) \leq \text{MaxTsValues} \\
\land \ \text{bootCtr} \leq \text{MaxBootCtr}
\]
C Proof of Initial Pasture

MODULE PastureNodeProof

EXTENDS PastureNode, TLAPS

STATE FUNCTIONS

We talk about the “log” being in various places. Actually, what is in those places is a cryptographic summary of the log, which is of type $Pcr$. However, under the anticollision assumption of $PcrExtend$, the cryptographic summary is effectively in one-to-one correspondance with the actual log. So we talk as if the cryptographic summary were the log, rather than merely a reference to the log.

Check that $ts$ is a valid seal attestation in the current node state. To be valid it must be a signed attestation and it must record $SemHappy$, $SealReboot$ and the current boot counter.

$$CheckTsIsCurrent(ts) \triangleq \land ts \in SignedTs$$
$$\land ts.semPcr = SemHappy$$
$$\land ts.sealPcr = SealReboot$$
$$\land ts.bootCtr = bootCtr$$

All valid seal attestations in the current node state. Seal attestations can be found among the known values (in $tvalues$) or in the temporary state variable $chkptts$ used during the checkpoint sem routine.

$$AllCurrentTs \triangleq \{ ts \in tvalues \cup \{ chkptts \} : CheckTsIsCurrent(ts) \}$$

If there are any valid seal attestations in the current node state, choose one and get its log.

$$CurrentTsLog \triangleq LET ts \triangleq CHOOSE ts \in AllCurrentTs : TRUE IN ts.appPcr$$

The log is present in the $nv$ ram. This is true iff the $nv$ ram says it is current.

$$LogInNv \triangleq \land nv.current$$

The log is present in the application $pcr$. This is true iff the sem $pcr$ contains $SemHappy$ and the seal $pcr$ contains $SealReboot$.

$$LogInApp \triangleq \land semPcr = SemHappy$$
$$\land sealPcr = SealReboot$$
The log is present in some known seal attestation. This is true iff there exists a valid seal attestation in the current node state.

\[ \text{LogInTs} \triangleq \text{AllCurrentTs} \neq \{\} \]

Assuming the log exists, determine if \( \text{Per p} \) is on it.

The log has a domicile in the \( \text{nq} \) ram, when the \( \text{nq} \) ram is marked as current. The log has a domicile in the application \( \text{pcr} \) when the sem \( \text{pcr} \) contains \( \text{SemHappy} \) and the seal \( \text{pcr} \) contains \( \text{SealReboot} \). The log has a domicile in a seal \( \text{ts} \) attestation when that attestation quotes \( \text{SemHappy} \), \( \text{SealReboot} \), and the current \( \text{bootCtr} \).

During secure execution mode, the log can also temporarily live in certain places, as it is moved from one domicile to another.

\[ \text{IsOnLog}(p) \triangleq \]

\[ \land \quad \text{LogInNu} \Rightarrow \text{PcrLeq}(p, \text{nv.appPcr}) \]
\[ \land \quad \text{LogInApp} \Rightarrow \text{PcrLeq}(p, \text{appPcr}) \]
\[ \land \quad \text{LogInTs} \Rightarrow \text{PcrLeq}(p, \text{CurrentTsLog}) \]

\[ \text{Special places to find the log during secure execution mode.} \]
\[ \land \quad \text{pc} = \text{PcRECOV1} \Rightarrow \text{TRUE} \]
\[ \land \quad \text{pc} = \text{PcRECOV2} \Rightarrow \text{PcrLeq}(p, \text{appPcr}) \]
\[ \land \quad \text{pc} = \text{PcRECOV3} \Rightarrow \text{PcrLeq}(p, \text{appPcr}) \]
\[ \land \quad \text{pc} = \text{PcCHKPT1} \Rightarrow \text{TRUE} \]
\[ \land \quad \text{pc} = \text{PcCHKPT2} \Rightarrow \text{PcrLeq}(p, \text{chkptts.appPcr}) \]
\[ \land \quad \text{pc} = \text{PcCHKPT3} \Rightarrow \text{PcrLeq}(p, \text{appPcr}) \]
\[ \land \quad \text{pc} = \text{PcCHKPT4} \Rightarrow \text{PcrLeq}(p, \text{nv.appPcr}) \]
\[ \land \quad \text{pc} = \text{PcCHKPT5} \Rightarrow \text{PcrLeq}(p, \text{nv.appPcr}) \]

**ADDITIONAL INVARIANTS**

When the node is in secure execution mode, the secure execution mode \( \text{pcr} \) contains \( \text{SemProtect} \).

\[ \text{InvInSemProtect} \triangleq \]
\[ \land \quad \text{InvType} \]
\[ \land \quad \text{goal::} \]
\[ \text{InSem} \Rightarrow \text{semPcr} = \text{SemProtect} \]

When the node is not in secure execution mode, the secure execution mode \( \text{pcr} \) contains a value from which \( \text{SemProtect} \) cannot be reached.

\[ \text{InvUnreachableSemProtect} \triangleq \]
\[ \land \quad \text{InvType} \]
\[ \land \quad \text{InvInSemProtect} \]
\[ \wedge \text{goal}:: \]
\[ \neg \text{InSem} \Rightarrow \neg \text{PcrLeq}(\text{semPcr}, \text{SemProtect}) \]

All known signed seal attestations quote a \textit{bootCtr} that does not exceed the current \textit{bootCtr}.

\[
\text{InvSignedTsLeqBoot} \triangleq \\
\wedge \text{InvType} \\
\wedge \text{goal}:: \\
\forall \text{ts} \in \text{tsvalues} \cup \{\text{chkptts}\} : \\
\text{ts} \in \text{SignedTs} \Rightarrow \text{ts.bootCtr} \leq \text{bootCtr}
\]

When the node is not in secure execution mode, the secure execution mode pcr contains either (1) \textit{SemHappy} or (2) a value from which \textit{SemHappy} cannot be reached.

\[
\text{InvUnforgeableSemHappy} \triangleq \\
\wedge \text{InvType} \\
\wedge \text{InvInSemProtect} \\
\wedge \text{goal}:: \\
\neg \text{InSem} \Rightarrow \\
\vee \text{semPcr} = \text{SemHappy} \\
\vee \neg \text{PcrLeq}(\text{semPcr}, \text{SemHappy})
\]

The seal pcr contains either (1) \textit{SealReboot} or (2) a value from which \textit{SealReboot} cannot be reached.

\[
\text{InvUnforgeableSealReboot} \triangleq \\
\wedge \text{InvType} \\
\wedge \text{goal}:: \\
\vee \text{sealPcr} = \text{SealReboot} \\
\vee \neg \text{PcrLeq}(\text{sealPcr}, \text{SealReboot})
\]

Every entry in obtains and revokes has a last extension of \textit{OBTAIN} and \textit{REVOKE}, respectively.

\[
\text{InvProperLastExtension} \triangleq \\
\wedge \text{InvType} \\
\wedge \text{goal}:: \\
\wedge \forall \text{o} \in \text{obtains} : \text{PcrHasExtension}(\text{o}) \wedge \text{PcrLastExtension}(\text{o}) = \text{PcrxOBTAIN} \\
\wedge \forall \text{r} \in \text{revokes} : \text{PcrHasExtension}(\text{r}) \wedge \text{PcrLastExtension}(\text{r}) = \text{PcrxREVOKE}
\]
There is at most one log.

One Log to rule them all,
One Log to find them,
One Log to bring them all
and in the darkness bind them.
(with apologies to J. R. R. Tolkien)

\[ InvOneLog \triangleq \]
\[ \land InvType \]
\[ \land InvSignedTsLeqBoot \]
\[ \land InvInSemProtect \]
\[ \land InvUnforgeableSemHappy \]
\[ \land InvUnforgeableSealReboot \]
\[ \land InvProperLastExtension \]
\[ \land goal:: \]
\[ The \ log \ can \ only \ have \ at \ most \ one \ domicile \ at \ a \ time. \]
\[ \land LogInNu \Rightarrow \neg LogInApp \land \neg LogInTs \]
\[ \land LogInApp \Rightarrow \neg LogInNu \land \neg LogInTs \]
\[ \land LogInTs \Rightarrow \neg LogInNu \land \neg LogInApp \]
Extra requirements during secure execution mode.
\[ \land pc = PcRECOV1 \Rightarrow TRUE \]
\[ \land pc = PcRECOV2 \Rightarrow LogInNu \]
\[ \land pc = PcRECOV3 \Rightarrow \neg LogInNu \land \neg LogInApp \land \neg LogInTs \]
\[ \land pc = PcCHKPT1 \Rightarrow TRUE \]
\[ \land pc = PcCHKPT2 \Rightarrow LogInTs \land CheckTsIsCurrent(chkptts) \]
\[ \land pc = PcCHKPT3 \Rightarrow LogInTs \land CheckTsIsCurrent(chkptts) \]
\[ \land pc = PcCHKPT4 \Rightarrow \neg LogInNu \land \neg LogInApp \land \neg LogInTs \]
\[ \land pc = PcCHKPT5 \Rightarrow LogInNu \]
All seal attestations containing the log must have the same log.
\[ \land \forall ts1, ts2 \in AllCurrentTs : ts1.appPcr = ts2.appPcr \]
Every entry in obtains (a decision to obtain access) is recorded on the log (assuming there is one).
\[ \land obtains:: \forall o \in obtains : IsOnLog(o) \]
Every entry in revokes (a decision to prove revocation) is recorded on the log (assuming there is one).
\[ \land revokes:: \forall r \in revokes : IsOnLog(r) \]
We have verifiable revocation.
\[ \land InvVerifiableRevocation \]

NECESSARY FACTS ABOUT NATURALS
The SMT prover can prove these easily enough in isolation, but if you ask it to prove them in the middle of other proofs where records and other complicated things are flying around, it usually aborts with a type inference failure.
\( \leq \) is a total order

THEOREM ThmNatLeqIsTotal \( \triangleq \forall i, j \in \mathbb{Nat} : i \leq j \lor j \leq i \) BY SMT

THEOREM ThmNatLeqIsReflexive \( \triangleq \forall i \in \mathbb{Nat} : i \leq i \) BY SMT

THEOREM ThmNatLeqIsAntisymmetric \( \triangleq \forall i, j \in \mathbb{Nat} : i \leq j \land j \leq i \Rightarrow i = j \) BY SMT

THEOREM ThmNatLeqIsTransitive \( \triangleq \forall i, j, k \in \mathbb{Nat} : i \leq j \land j \leq k \Rightarrow i \leq k \) BY SMT

\( \leq \) minimum is 0

THEOREM ThmNatLeqMinIsZero \( \triangleq \forall i \in \mathbb{Nat} : 0 \leq i \) BY SMT

\( \leq \) is the opposite of >

THEOREM ThmNatLeqXorGt \( \triangleq \forall i, j \in \mathbb{Nat} : i \leq j \equiv \neg (i > j) \) BY SMT

THEOREM ThmNatMore \( \triangleq \forall i, j \in \mathbb{Nat} : i \leq i + j \) BY SMT

THEOREM ThmNatLess \( \triangleq \forall i, j \in \mathbb{Nat} : i - j \leq i \) BY SMT

THEOREM ThmNatInc \( \triangleq \forall i \in \mathbb{Nat} : i + 1 > i \) BY SMT

THEOREM ThmNatDotDot \( \triangleq \forall i, j, k \in \mathbb{Nat} : i \leq j \land j \leq k \equiv j \in i \ldots k \) BY SMT

THEOREM ThmNatDecZero \( \triangleq \forall n \in \mathbb{Nat} : n > 0 \Rightarrow n - 1 \in \mathbb{Nat} \) BY SMT

THEOREM ThmNatAddEq \( \triangleq \forall i, j, k \in \mathbb{Nat} : i + k = j + k \Rightarrow i = j \) BY SMT

THEOREM ThmNatLeqLt \( \triangleq \forall i, j, k \in \mathbb{Nat} : i \leq j \land j < k \Rightarrow i < k \) BY SMT

NECESSARY FACTS ABOUT SEQUENCES

I have not been able to figure out how to convince the prover to prove most of these.

Definition of a sequence.

THEOREM ThmSeqDef \( \triangleq \)

ASSUME

NEW CONSTANT \( S \),
NEW CONSTANT \( q \in \text{Seq}(S) \)

PROVE

\[ q = [i \in 1.. \text{Len}(q) \mapsto q[i]] \]

PROOF

OMITTED

The empty sequence is a sequence of \( S \), for any \( S \).
THEOREM ThmSeqEmptyIsSeq $\triangleq$
ASSUME
NEW CONSTANT $S$
PROVE
$\langle \rangle \in Seq(S)$
PROOF
OMITTED

For any sequence $q$ of $S$, $\text{Len}(q) \in \text{Nat}$.

THEOREM ThmSeqLenIsNat $\triangleq$
ASSUME
NEW CONSTANT $S$
NEW $q \in Seq(S)$
PROVE
$\text{Len}(q) \in \text{Nat}$
PROOF
OMITTED

For any non-empty sequence of $S$, its tail is a sequence of $S$.

THEOREM ThmSeqTailIsSeq $\triangleq$
ASSUME
NEW CONSTANT $S$
NEW CONSTANT $q \in Seq(S)$
$q \neq \langle \rangle$
PROVE
$\text{Tail}(q) \in Seq(S)$
PROOF
OMITTED

For any sequence of $S$, appending $x \in S$ yields a sequence of $S$.

THEOREM ThmSeqAppendIsSeq $\triangleq$
ASSUME
NEW CONSTANT $S$
NEW CONSTANT $q \in Seq(S)$
NEW CONSTANT $x \in S$
PROVE
$\text{Append}(q, x) \in Seq(S)$
PROOF
OMITTED

The result of $\text{Append}(q, x)$ is one longer than $q$. 
THEOREM ThmSeqAppendLen1 \(\triangleq\)

ASSUME
- NEW CONSTANT \(S\),
- NEW CONSTANT \(q \in \text{Seq}(S)\),
- NEW CONSTANT \(x \in S\)

PROVE
\[\text{Len(Append}(q, x)) = \text{Len}(q) + 1\]

PROOF
OMITTED

The result of \(\text{Append}(q, x)\) starts with \(q\).

THEOREM ThmSeqAppendSubSeq \(\triangleq\)

ASSUME
- NEW CONSTANT \(S\),
- NEW CONSTANT \(q \in \text{Seq}(S)\),
- NEW CONSTANT \(x \in S\)

PROVE
\[\text{SubSeq(Append}(q, x), 1, \text{Len}(q)) = q\]

PROOF
OMITTED

Appending the last entry onto all but the last of a sequence yields the original sequence.

THEOREM ThmSeqAppendPriorLast \(\triangleq\)

ASSUME
- NEW CONSTANT \(S\),
- NEW CONSTANT \(q \in \text{Seq}(S)\),
- \(\text{Len}(q) > 0\)

PROVE
\[\text{Append(SubSeq}(q, 1, \text{Len}(q) - 1), q[\text{Len}(q)]) = q\]

PROOF
OMITTED

The entire initial \(\text{SubSeq}\) of \(q\) is \(q\).

THEOREM ThmSeqEntireInitialSubSeq \(\triangleq\)

ASSUME
- NEW CONSTANT \(S\),
- NEW CONSTANT \(q \in \text{Seq}(S)\)

PROVE
\[q = \text{SubSeq}(q, 1, \text{Len}(q))\]

PROOF
OMITTED
Initial $\text{SubSeq} \in \text{sequence}$.

**THEOREM** $\text{ThmSeqInitialSubSeqIsEq}$

**ASSUME**
- NEW CONSTANT $S$,
- NEW CONSTANT $q \in \text{Seq}(S)$,
- NEW CONSTANT $n \in \text{Nat}$,
  - $n \leq \text{Len}(q)$

**PROVE**
- $\text{SubSeq}(q, 1, n) \in \text{Seq}(S)$

**PROOF**
OMITTED

Initial $\text{SubSeq}$ is antisymmetric.

**THEOREM** $\text{ThmSeqInitialSubSeqIsAntisymmetric}$

**ASSUME**
- NEW CONSTANT $S$,
- NEW CONSTANT $q \in \text{Seq}(S)$,
- NEW CONSTANT $r \in \text{Seq}(S)$,
  - $\text{Len}(q) \leq \text{Len}(r)$,
  - $\text{Len}(r) \leq \text{Len}(q)$,
  - $q = \text{SubSeq}(r, 1, \text{Len}(q))$,
  - $r = \text{SubSeq}(q, 1, \text{Len}(r))$

**PROVE**
- $q = r$

**PROOF**
 1. $\text{Len}(q) = \text{Len}(r)$
 2. USE $\text{ThmSeqLenIsNat}$
 3. (2) QED BY $\text{ThmNatLeqIsAntisymmetric}$
 4. (1) QED BY $\text{ThmSeqEntireInitialSubSeq}$

Initial $\text{SubSeq}$ is transitive.

**THEOREM** $\text{ThmSeqInitialSubSeqIsTransitive}$

**ASSUME**
- NEW CONSTANT $S$,
- NEW CONSTANT $q \in \text{Seq}(S)$,
- NEW CONSTANT $r \in \text{Seq}(S)$,
- NEW CONSTANT $s \in \text{Seq}(S)$,
  - $\text{Len}(q) \leq \text{Len}(r)$,
  - $\text{Len}(r) \leq \text{Len}(s)$,
  - $q = \text{SubSeq}(r, 1, \text{Len}(q))$,
  - $r = \text{SubSeq}(s, 1, \text{Len}(r))$

**PROVE**
\[ q = \text{SubSeq}(s, 1, \text{Len}(q)) \]

PROOF
OMITTED

---

### Sequence append incompatible.

**THEOREM** \textit{ThmSeqAppendIncompatible} \[\triangleq\]

**ASSUME**

- NEW CONSTANT \( S \),
- NEW CONSTANT \( q \in \text{Seq}(S) \),
- NEW CONSTANT \( s_1 \in S \),
- NEW CONSTANT \( s_2 \in S \),
- \( s_1 \neq s_2 \)

**PROVE**

\( \text{Append}(q, s_1) \neq \text{Append}(q, s_2) \)

PROOF
OMITTED

---

### Sequence append anti-collision.

**THEOREM** \textit{ThmSeqAppendAnticollision} \[\triangleq\]

**ASSUME**

- NEW CONSTANT \( S \),
- NEW CONSTANT \( q \),
- NEW CONSTANT \( q_1 \in \text{Seq}(S) \),
- NEW CONSTANT \( q_2 \in \text{Seq}(S) \),
- NEW CONSTANT \( s_1 \in S \),
- NEW CONSTANT \( s_2 \in S \),
- \( q = \text{Append}(q_1, s_1) \),
- \( q = \text{Append}(q_2, s_2) \)

**PROVE**

\( q_1 = q_2 \land s_1 = s_2 \)

PROOF
OMITTED
\( \text{PcrInit} \in \text{Per} \)

**THEOREM ThmPcrInitIsPcr**
\[
\forall i \in \text{Pcri} : \text{PcrInit}(i) \in \text{Pcr}
\]

**PROOF**

1. TAKE \( i \in \text{Pcri} \)
2. USE DEF \( \text{Pcr}, \text{PcrInit} \)
3. DEFINE \( p \triangleq \text{PcrInit}(i) \)
4. \( \text{p.init} \in \text{Pcri} \) OBVIOUS
5. \( \text{p.extq} \in \text{Seq}({\text{Pcrx}}) \) BY ThmSeqEmptyIsSeq
6. QED BY (1)1, (1)2

\( \text{PcrExtend} \in \text{Pcr} \)

**THEOREM ThmPcrExtendIsPcr**
\[
\forall p \in \text{Per}, x \in \text{Pcrx} : \text{PcrExtend}(p, x) \in \text{Pcr}
\]

**PROOF**

1. TAKE \( p \in \text{Per}, x \in \text{Pcrx} \)
2. USE DEF \( \text{Pcr}, \text{PcrExtend} \)
3. DEFINE \( px \triangleq \text{PcrExtend}(p, x) \)
4. \( \text{px.init} \in \text{Pcri} \) OBVIOUS
5. \( \text{px.extq} \in \text{Seq}({\text{Pcrx}}) \) BY ThmSeqAppendIsSeq
6. QED BY (1)1, (1)2

\( \text{PcrLen} \in \text{Nat} \)

**THEOREM ThmPcrLenIsNat**
\[
\forall p \in \text{Pcr} : \text{PcrLen}(p) \in \text{Nat}
\]

**PROOF**

1. TAKE \( p \in \text{Pcr} \)
2. USE DEF \( \text{PcrLen} \)
3. USE DEF \( \text{Pcr} \)
4. QED BY ThmSeqLenIsNat

\( p \leq \text{PcrExtend}(p, x) \)

**THEOREM ThmPcrExtendLeq**
\[
\forall p \in \text{Pcr}, x \in \text{Pcrx} : \text{PcrLeq}(p, \text{PcrExtend}(p, x))
\]

**PROOF**

1. TAKE \( p \in \text{Pcr}, x \in \text{Pcrx} \)
2. USE DEF \( px \triangleq \text{PcrExtend}(p, x) \)
3. USE DEF \( \text{Pcr} \)
4. USE DEF \( \text{PcrExtend} \)
5. USE DEF \( \text{PcrLeq} \)
6. \( \text{p.init} = \text{px.init} \) OBVIOUS
7. \( \text{Len}(p.extq) \leq \text{Len}(px.extq) \)
\[\text{(2) } \text{Len}(p\text{.extq}) + 1 = \text{Len}(px\text{.extq}) \text{ BY } \text{ThmSeqAppendLen}\]
\[\text{(2) USE } \text{ThmSeqLenIsNat}\]
\[\text{(2) QED BY } \text{ThmNatMore}\]

\(p \neq \text{PcrExtend}(p, x)\)

\textbf{THEOREM ThmPcrExtendNeq} \(\overset{\Delta}{=}\)

\[\forall p \in \text{Pcr}, x \in \text{Pcrx} : p \neq \text{PcrExtend}(p, x)\]

\textbf{PROOF}

(1) \text{TAKLE } p \in \text{Pcr}, x \in \text{Pcrx}

(1) \text{DEFINE } px \overset{\Delta}{=} \text{PcrExtend}(p, x)

(1) \text{USE DEF Pcr}

(1) \text{USE DEF PcrExtend}

(1) \text{p.extq} \neq \text{px.extq}

(2) \text{p.extq} \in \text{Seq(Pcrx)} \text{ OBVIOUS}

(2) \text{px.extq} \in \text{Seq(Pcrx)} \text{ BY } \text{ThmSeqAppendIsSeq}

(2) \text{DEFINE } pn \overset{\Delta}{=} \text{Len}(p\text{.extq})

(2) \text{DEFINE } pxn \overset{\Delta}{=} \text{Len}(px\text{.extq})

(2) \text{pn} \neq \text{pxn}

(3) \text{pn} \in \text{Nat} \text{ BY } \text{ThmSeqLenIsNat}

(3) \text{pxn} \in \text{Nat} \text{ BY } \text{ThmSeqLenIsNat}

(3) \text{pxn} = \text{pn} + 1 \text{ BY } \text{ThmSeqAppendLen}\overset{1}{1}

(3) \text{pxn} > \text{pn} \text{ BY } \text{ThmNatInc}

(3) \neg(\text{pxn} \leq \text{pn}) \text{ BY } \text{ThmNatLeqXorGt}

(3) \text{QED BY } \text{ThmNatLeqIsReflexive}

(2) \text{QED OBVIOUS}

(1) \text{QED OBVIOUS}

\text{Pcr equality. This would seem to be trivial but the prover cannot seem to figure it out by itself.}

\textbf{THEOREM ThmPcrEqual} \(\overset{\Delta}{=}\)

\[\forall p, q \in \text{Pcr} :\]

\[\land \ p\text{.init} = q\text{.init}\]

\[\land \ p\text{.extq} = q\text{.extq}\]

\[\Rightarrow p = q\]

\textbf{PROOF}

(1) \text{TAKLE } p, q \in \text{Pcr}

(1) \text{HAVE } p\text{.init} = q\text{.init} \land p\text{.extq} = q\text{.extq}

(1) \text{USE DEF Pcr}

The following fact seems to be necessary to help the prover.

(1) \text{p = [q EXCEPT !.init = p.init, !.extq = p.extq]OBVIOUS}

(1) \text{QED OBVIOUS}
Anti-collision property.

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

\[ \forall p_1, p_2 \in \text{Pcr}, x_1, x_2 \in \text{Pcr} : \]

\[ \text{PcrExtend}(p_1, x_1) = \text{PcrExtend}(p_2, x_2) \Rightarrow p_1 = p_2 \land x_1 = x_2 \]

**PROOF**

(1) TAKE \( p_1, p_2 \in \text{Pcr}, x_1, x_2 \in \text{Pcr} \)

(1) DEFINE \( px_1 \triangleq \text{PcrExtend}(p_1, x_1) \)

(1) DEFINE \( px_2 \triangleq \text{PcrExtend}(p_2, x_2) \)

(1) HAVE \( px_1 = px_2 \)

(1) USE DEF \( \text{Pcr} \)

(1) USE DEF \( \text{PcrExtend} \)

(1) QED

(2) \( p_1.\text{init} = p_2.\text{init} \) OBVIOUS

(2) \( p_1.\text{extq} = p_2.\text{extq} \land x_1 = x_2 \)

Create definitions for the \( \text{extq} \) fields and then hide them to prevent overwhelming the prover.

(3) DEFINE \( p_1q \triangleq p_1.\text{extq} \)

(3) DEFINE \( p_2q \triangleq p_2.\text{extq} \)

(3) HIDE DEF \( p_1q \)

(3) HIDE DEF \( p_2q \)

(3) \( \text{Append}(p_1q, x_1) = \text{Append}(p_2q, x_2) \) BY DEF \( p_1q, p_2q \)

(3) \( p_1q \in \text{Seq} (\text{Pcr}) \) BY DEF \( p_1q \)

(3) \( p_2q \in \text{Seq} (\text{Pcr}) \) BY DEF \( p_2q \)

(3) \( p_1q = p_2q \land x_1 = x_2 \) BY \( \text{ThmSeqAppendAnticollision} \)

(3) QED BY DEF \( p_1q, p_2q \)

(2) QED BY \( \text{ThmPcrEqual} \)

If two extensions of the same pcr are both \( \leq \) a target pcr, then the extensions must be the same.

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

\[ \forall p, t \in \text{Pcr}, x_1, x_2 \in \text{Pcr} : \]

\[ \text{PcrLeq}(\text{PcrExtend}(p, x_1), t) \land \text{PcrLeq}(\text{PcrExtend}(p, x_2), t) \Rightarrow x_1 = x_2 \]

**PROOF**

(1) TAKE \( p, t \in \text{Pcr}, x_1, x_2 \in \text{Pcr} \)

(1) HAVE \( \text{PcrLeq}(\text{PcrExtend}(p, x_1), t) \land \text{PcrLeq}(\text{PcrExtend}(p, x_2), t) \)

(1) USE DEF \( \text{Pcr} \)

(1) USE DEF \( \text{PcrExtend} \)

(1) USE DEF \( \text{PcrLeq} \)

(1) DEFINE \( qp \triangleq p.\text{extq} \)

(1) DEFINE \( qt \triangleq t.\text{extq} \)

(1) DEFINE \( qpx_1 \triangleq \text{Append}(qp, x_1) \)

(1) DEFINE \( qpx_2 \triangleq \text{Append}(qp, x_2) \)

(1) \( qp \in \text{Seq} (\text{Pcr}) \) OBVIOUS

(1) \( qt \in \text{Seq} (\text{Pcr}) \) OBVIOUS

(1) \( \text{Len}(qpx_1) \leq \text{Len}(qt) \) OBVIOUS

(1) \( \text{Len}(qpx_2) \leq \text{Len}(qt) \) OBVIOUS
\( (1) \) \text{SubSeq}(qt, 1, \text{Len}(qpx_1)) = qpx_1 \text{OBVIOUS}

\( (1) \) \text{SubSeq}(qt, 1, \text{Len}(qpx_2)) = qpx_2 \text{OBVIOUS}

\( (1) \) HIDE DEF \( qp \)

\( (1) \) HIDE DEF \( qt \)

\( (1) \) \text{Len}(qpx_1) = \text{Len}(qpx_2) \text{OBVIOUS}

\( (1) \) \text{qpx}_1 = \text{qpx}_2 \text{OBVIOUS}

\( (1) \) QED

\begin{verbatim}
The prover really needs help to focus its attention.
\end{verbatim}

\( ⟨2⟩1. \ x_1 \in Pcr \text{OBVIOUS} \\
⟨2⟩2. \ x_2 \in Pcr \text{OBVIOUS} \\
⟨2⟩3. \ qp \in \text{Seq}(Pcr) \text{OBVIOUS} \\
⟨2⟩4. \ \text{Append}(qp, x_1) = \text{Append}(qp, x_2) \text{OBVIOUS} \\
⟨2⟩ QED BY ONLY \( ⟨2⟩1, ⟨2⟩2, ⟨2⟩3, ⟨2⟩4, \text{ThmSeqAppendAnticollision} \)

\begin{verbatim}
PcrExtend increases the length by 1.

THEOREM ThmPcrExtendLen \( \triangleq \)
\[ \forall p \in Pcr, x \in Pcrx : \text{PcrLen}(\text{PcrExtend}(p, x)) = \text{PcrLen}(p) + 1 \]

PROOF
\( (1) \) TAKE \( p \in Pcr, x \in Pcrx \\
\( (1) \) DEFINE \( px \triangleq \text{PcrExtend}(p, x) \\
\( (1) \) \( px \in Pcr \text{BY ThmPcrExtendIsPcr} \\
\( (1) \) USE DEF \text{PcrLen} \\
\( (1) \) USE DEF \text{PcrExtend} \\
\( (1) \) USE DEF \text{Pcr} \\
\( (1) \) QED BY \text{ThmSeqAppendLen} \( 1 \)
\end{verbatim}

\begin{verbatim}
PcrLeq implies \( \leq \) on respective PcrLen.

THEOREM ThmPcrLeqLeq \( \triangleq \)
\[ \forall p, q \in Pcr : \text{PcrLeq}(p, q) \Rightarrow \text{PcrLen}(p) \leq \text{PcrLen}(q) \]

PROOF
\( (1) \) TAKE \( p, q \in Pcr \\
\( (1) \) HAVE \text{PcrLeq}(p, q) \\
\( (1) \) USE DEF \text{Pcr} \\
\( (1) \) \text{Len}(p.{ext}q) \leq \text{Len}(q.{ext}q) \text{BY DEF PcrLeq} \\
\( (1) \) \text{PcrLen}(p) = \text{Len}(p.{ext}q) \text{BY DEF PcrLen} \\
\( (1) \) \text{PcrLen}(q) = \text{Len}(q.{ext}q) \text{BY DEF PcrLen} \\
\( (1) \) QED OBVIOUS
\end{verbatim}
PROOF OF INITIAL PASTURE

\textit{PcrLeq} is a partial order.

THEOREM \textit{ThmPcrLeqIsReflexive} \(\triangleq\)
\[
\forall p \in \text{Pcr} : \text{PcrLeq}(p, p)
\]

PROOF
\(\langle 1 \rangle\) \textbf{TAKE} \(p \in \text{Pcr}\)
\(\langle 1 \rangle\) \textbf{USE} \textit{DEF} \textit{PcrLeq}
\(\langle 1 \rangle\) \textbf{USE} \textit{DEF} \textit{Pcr}
\(\langle 1 \rangle\) \textit{1.} \(p\text{.init} = p\text{.init}\) \textbf{OBVIOUS}
\(\langle 1 \rangle\) \textit{2.} \(\text{Len}(p\text{.extq}) \leq \text{Len}(p\text{.extq})\)
\(\langle 2 \rangle\) \textbf{USE} \textit{ThmSeqLenIsNat}
\(\langle 2 \rangle\) \textbf{USE} \textit{ThmNatLeqIsReflexive}
\(\langle 2 \rangle\) \textbf{QED} \textbf{OBVIOUS}
\(\langle 1 \rangle\) \textit{3.} \(p\text{.extq} = \text{SubSeq}(p\text{.extq}, 1, \text{Len}(p\text{.extq}))\)
\(\langle 2 \rangle\) \textbf{USE} \textit{ThmSeqEntireInitialSubSeq}
\(\langle 2 \rangle\) \textbf{QED} \textbf{OBVIOUS}
\(\langle 1 \rangle\) \textbf{QED BY} \(\langle 1 \rangle\)\textit{1}, \(\langle 1 \rangle\)\textit{2}, \(\langle 1 \rangle\)\textit{3}

THEOREM \textit{ThmPcrLeqIsAntisymmetric} \(\triangleq\)
\[
\forall p, q \in \text{Pcr} : \text{PcrLeq}(p, q) \land \text{PcrLeq}(q, p) \Rightarrow p = q
\]

PROOF
\(\langle 1 \rangle\) \textbf{TAKE} \(p, q \in \text{Pcr}\)
\(\langle 1 \rangle\) \textbf{HAVE} \textit{PcrLeq}(p, q) \land \textit{PcrLeq}(q, p)
\(\langle 1 \rangle\) \textit{3.} \(p\text{.init} = q\text{.init}\)
\(\langle 2 \rangle\) \textbf{USE} \textit{DEF} \textit{PcrLeq}
\(\langle 2 \rangle\) \textbf{QED BY} \(\langle 1 \rangle\)\textit{2}
\(\langle 1 \rangle\) \textit{4.} \(p\text{.extq} = q\text{.extq}\)
\(\langle 2 \rangle\) \textbf{USE} \textit{DEF} \textit{PcrLeq}
\(\langle 2 \rangle\) \textbf{USE} \textit{DEF} \textit{Pcr}
\(\langle 2 \rangle\) \textbf{USE} \textit{ThmSeqInitialSubSeqIsAntisymmetric}
\(\langle 2 \rangle\) \textbf{QED BY} \(\langle 1 \rangle\)\textit{2}
\(\langle 1 \rangle\) \textbf{QED}
\(\langle 2 \rangle\) \textbf{USE} \textit{ThmPcrEqual}
\(\langle 2 \rangle\) \textbf{QED BY} \(\langle 1 \rangle\)\textit{3}, \(\langle 1 \rangle\)\textit{4}

THEOREM \textit{ThmPcrLeqIsTransitive} \(\triangleq\)
\[
\forall p, q, r \in \text{Pcr} : \text{PcrLeq}(p, q) \land \text{PcrLeq}(q, r) \Rightarrow \text{PcrLeq}(p, r)
\]

PROOF
\(\langle 1 \rangle\) \textbf{TAKE} \(p, q, r \in \text{Pcr}\)
\(\langle 1 \rangle\) \textbf{HAVE} \textit{PcrLeq}(p, q) \land \textit{PcrLeq}(q, r)
\(\langle 1 \rangle\) \textbf{USE} \textit{DEF} \textit{PcrLeq}
\(\langle 1 \rangle\) \textit{1.} \(p\text{.init} = r\text{.init}\) \textbf{OBVIOUS}
\(\langle 1 \rangle\) \textit{2.} \(\text{Len}(p\text{.extq}) \leq \text{Len}(r\text{.extq})\)
\(\langle 2 \rangle\) \textbf{USE} \textit{DEF} \textit{Pcr}
\(\langle 2 \rangle\) \textbf{USE} \textit{ThmSeqLenIsNat}
An extension of a Pcr $p$ cannot reach $p$.

**THEOREM ThmPcrExtendSelfUnreachable**

\[ \forall p \in \text{Pcr}, x \in \text{Pcr} : \neg \text{PcrLeq(PcrExtend}(p, x), p) \]

**PROOF**

1. TAKE $p \in \text{Pcr}, x \in \text{Pcr}$
2. DEFINE $px \triangleq \text{PcrExtend}(p, x)$
3. DEFINE $isleq \triangleq \text{PcrLeq}(px, p)$
4. DEFINE $px \in \text{Pcr}$ BY ThmPcrExtendIsPcr

Proof by contradiction.

1. CASE $\neg isleq$ BY (1)1
2. CASE $isleq$
   1. $\text{PcrLen}(px) \leq \text{PcrLen}(p)$
      2. USE (1)2
      3. USE ThmPcrLeqLeq
      4. QED OBVIOUS
   2. $\text{PcrLen}(px) > \text{PcrLen}(p)$
      3. USE ThmPcrExtendLen
      4. QED OBVIOUS

1. USE ThmPcrLenIsNat
2. USE ThmNatInc
3. QED OBVIOUS

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

If an extension of a Pcr can reach a target, the Pcr itself can reach the target.

**THEOREM ThmPcrReachableIfExtend**

\[ \forall p, q \in \text{Pcr}, x \in \text{Pcr} : \text{PcrLeq(PcrExtend}(p, x), q) \Rightarrow \text{PcrLeq}(p, q) \]

**PROOF**

1. TAKE $p, q \in \text{Pcr}, x \in \text{Pcr}$
2. DEFINE $px \triangleq \text{PcrExtend}(p, x)$
3. HAVE $\text{PcrLeq}(px, q)$
4. DEFINE $px \in \text{Pcr}$ BY ThmPcrExtendIsPcr
5. $\text{PcrLeq}(p, px)$ BY ThmPcrExtendLeq
6. QED BY ThmPcrLeqIsTransitive
If a target \( Pcr \) is not reachable from a source \( Pcr \), then it is not reachable from an extension of the source \( Pcr \).

**THEOREM** \( \text{ThmPcrExtendSourceUnreachable} \) $\triangleq$
\[
\forall p, q \in Pcr, x \in Pcrx : \\
\neg PcrLeq(p, q) \Rightarrow \neg PcrLeq(PcrExtend(p, x), q)
\]

**PROOF**
1. QED by \( \text{ThmPcrReachableIfExtend} \)

If \( p \) equals \( q \) or cannot reach \( q \), then an extension of \( p \) cannot reach \( q \).

**THEOREM** \( \text{ThmPcrExtendFromEqOrNotleq} \) $\triangleq$
\[
\forall p, q \in Pcr, x \in Pcrx : \\
p = q \lor \neg PcrLeq(p, q) \Rightarrow \neg PcrLeq(PcrExtend(p, x), q)
\]

**PROOF**
1. TAKE \( p, q \in Pcr, x \in Pcrx 
2. HAVE \( p = q \lor \neg PcrLeq(p, q) 
3. CASE \( p = q \) BY \( \text{ThmPcrExtendSelfUnreachable} \)
4. CASE \( \neg PcrLeq(p, q) \) BY \( \text{ThmPcrReachableIfExtend} \)
5. QED OBVIOUS

Different extensions of a \( Pcr \) are incompatible.

**THEOREM** \( \text{ThmPcrExtendIncompatible} \) $\triangleq$
\[
\forall p \in Pcr, x1, x2 \in Pcrx : \\
x1 \neq x2 \Rightarrow \neg PcrLeq(PcrExtend(p, x1), PcrExtend(p, x2))
\]

**PROOF**
1. TAKE \( p \in Pcr, x1, x2 \in Pcrx 
2. HAVE \( x1 \neq x2 
3. DEFINE \( p1 \triangleq PcrExtend(p, x1) 
4. DEFINE \( p2 \triangleq PcrExtend(p, x2) 
5. CASE \( \neg PcrLeq(p1, p2) \) BY (1) \( \bigtriangleup \)
6. CASE \( PcrLeq(p1, p2) \) \( \bigtriangleup \)
7. USE (1) \( \bigtriangleup \)
8. USE DEF \( PcrLeq 
9. USE DEF \( PcrExtend 
10. USE DEF \( Pcr 
11. p1.\text{extq} \in \text{Seq}(Pcrx) \) BY \( \text{ThmSeqAppendIsSeq} \)
12. p1.\text{extq} \in \text{Seq}(Pcrx) \) BY \( \text{ThmSeqAppendIsSeq} \)
13. \( \text{Len}(p1.\text{extq}) = \text{Len}(p2.\text{extq}) \) BY \( \text{ThmSeqAppendLen} \)
14. \( \text{Len}(p1.\text{extq}) = \text{Len}(p2.\text{extq}) + 1 \) BY \( \text{ThmSeqAppendLen} \)
15. \( \text{QED OBVIOUS} \)
16. CASE \( p1.\text{extq} \neq p2.\text{extq} \) BY \( \text{ThmSeqEntireInitialSubSeq} \)
17. CASE \( p1.\text{extq} \neq p2.\text{extq} \) BY \( \text{ThmSeqAppendIncompatible} \)
If a \( Pcr \) has an extension, applying \( PriorPcr \) to it yields a \( Pcr \).

**THEOREM** \( ThmPcrPriorIsPcr \) \( \triangleq \)
\[
\forall p \in Pcr : PcrHasExtension(p) \Rightarrow PcrPrior(p) \in Pcr
\]

**PROOF**

(1) TAKE \( p \in Pcr 
(1) HAVE \( PcrHasExtension(p) 
(1) USE DEF \( PcrHasExtension 
(1) USE DEF \( PcrPrior 
(1) USE DEF \( PcrLen 
(1) USE DEF \( Pcr 
(1) \text{DEFINE} \ p_0 \triangleq PcrPrior(p) 
(1) \text{DEFINE} \ x \triangleq PcrLastExtension(p) 
(1) \text{DEFINE} \ p_0, x.initial = p.initial 
(1) \text{DEFINE} \ x.extq = p.extq 
(2) \text{SUFFICES} 
ASSUME \( \text{Len}(qp) > 0, 

Putting the last extension back on the prior \( pcr \) yields the original \( pcr \).

**THEOREM** \( ThmPcrExtendPriorLast \) \( \triangleq \)
\[
\forall p \in Pcr : PcrHasExtension(p) \Rightarrow PcrExtend(PcrPrior(p), PcrLastExtension(p)) = p
\]

**PROOF**

(1) TAKE \( p \in Pcr 
(1) HAVE \( PcrHasExtension(p) 
(1) USE DEF \( PcrHasExtension 
(1) USE DEF \( PcrPrior 
(1) USE DEF \( PcrLastExtension 
(1) USE DEF \( PcrExtend 
(1) USE DEF \( PcrLen 
(1) USE DEF \( Pcr 
(1) DEFINE \ p_0 \triangleq PcrPrior(p) 
(1) DEFINE \ x \triangleq PcrLastExtension(p) 
(1) \text{DEFINE} \ p_0, x.initial = p.initial \text{OBVIOUS} 
(1) \text{DEFINE} \ x.extq = p.extq 
(2) \text{SUFFICES} 
ASSUME \( \text{Len}(qp) > 0, 

\( \langle 2 \rangle \) QED OBVIOUS 
(1) QED BY (1)1, (1)2
\( qp \in Seq(Pcrx) \)

**PROOF**

\[
\text{Append}(\text{SubSeq}(qp, 1, \text{Len}(qp) - 1), qp[\text{Len}(qp)]) = qp
\]

**OBVIOUS**

(2) *HIDE DEF* \( qp \)

(2) *QED BY ThmSeqAppendPriorLast*

(1) *QED BY ThmPcrEqual*

---

**WELL KNOWN PCR VALUES**

Value of the application pcr attained by rebooting.

**THEOREM** \( \text{ThmAppRebootIsPcr} \triangleq \text{AppReboot} \in Pcr \)

**PROOF**

(1) *USE DEF AppReboot*

(1) *USE DEF Pcri*

(1) *QED BY ThmPcrInitIsPcr*

Value of the secure execution mode pcr attained by rebooting.

**THEOREM** \( \text{ThmSemRebootIsPcr} \triangleq \text{SemReboot} \in Pcr \)

**PROOF**

(1) *USE DEF SemReboot*

(1) *USE DEF Pcri*

(1) *QED BY ThmPcrInitIsPcr*

Value of the secure execution mode pcr attained by entering the protected module in secure execution mode.

**THEOREM** \( \text{ThmSemProtectIsPcr} \triangleq \text{SemProtect} \in Pcr \)

**PROOF**

(1) *USE DEF SemProtect*

(1) *USE DEF Pcri*

(1) *QED BY ThmPcrInitIsPcr*

Value of the secure execution mode pcr that indicates that Pasture is happy. Recovery has been properly performed and bound keys may be used. Checkpoint has not yet been invoked.

**THEOREM** \( \text{ThmSemHappyIsPcr} \triangleq \text{SemHappy} \in Pcr \)

**PROOF**

(1) *USE DEF SemHappy*
Value of the seal pcr attained by rebooting.

THEOREM ThmSealRebootIsPcr \(\triangleq\) SealReboot \(\in\) Pcr

PROOF

(1) USE DEF SealReboot
(1) USE DEF PcrInit
(1) QED BY ThmPcrInitIsPcr

From SemReboot cannot reach SemProtect.

THEOREM ThmSemRebootNotleqSemProtect \(\triangleq\) \neg PcrLeq(SemReboot, SemProtect)

PROOF

(1) USE DEF PcrInit
(1) USE DEF SemReboot
(1) USE DEF SemProtect
(1) USE AssSemProtect
(1) QED BY DEF PcrLeq

PROTECTED NV RAM STATE

THEOREM ThmInitNvIsNv \(\triangleq\) InitNv \(\in\) Nv

PROOF

(1) USE DEF InitNv
(1) USE DEF Nv
(1) USE ThmAppRebootIsPcr
(1) QED OBVIOUS
SEAL OPERATION TRANSPORT SESSION STATE

THEOREM $\text{ThmNullTsIsTs} \triangleq \text{NullTs } \in \text{Ts}$

PROOF

(1) USE DEF $\text{Ts}$

(1) QED OBVIOUS

THEOREM $\text{ThmNullTsIsntSignedTs} \triangleq \text{NullTs } \notin \text{SignedTs}$

PROOF

(1) USE DEF $\text{NullTs}$

(1) USE $\text{NoSetContainsEverything}$

(1) QED OBVIOUS

PROOF OF INVARIANT $\text{InvType}$

It holds in the initial state.

THEOREM $\text{ThmInitInvType} \triangleq$

$\text{Init } \Rightarrow \text{InvType}$

PROOF

(1) HAVE $\text{Init}$

(1) USE DEF $\text{Init}$

(1) USE DEF $\text{Pc, PcRecover, PcCheckpoint}$

(1) USE DEF $\text{PcIDLE}$

(1) USE DEF $\text{PcRecover1, PcRecover2, PcRecover3}$

(1) USE DEF $\text{PcCheckpoint1, PcCheckpoint2, PcCheckpoint3, PcCheckpoint4, PcCheckpoint5}$

Just walk through each variable.

(1) $\text{InvType}!\text{nv} \quad \text{BY ThmInitNvIsNv}$

(1) $\text{InvType}!\text{appPcr} \quad \text{BY ThmAppRebootIsPcr}$

(1) $\text{InvType}!\text{semPcr} \quad \text{BY ThmSemRebootIsPcr}$

(1) $\text{InvType}!\text{sealPcr} \quad \text{BY ThmSealRebootIsPcr}$

(1) $\text{InvType}!\text{bootCdr} \quad \text{OBVIOUS}$

(1) $\text{InvType}!\text{pc} \quad \text{OBVIOUS}$

(1) $\text{InvType}!\text{chkptts} \quad \text{BY ThmNullTsIsTs}$

(1) $\text{InvType}!\text{tsvalues} \quad \text{OBVIOUS}$

(1) $\text{InvType}!\text{obtains} \quad \text{OBVIOUS}$

(1) $\text{InvType}!\text{revokes} \quad \text{OBVIOUS}$

(1) QED BY DEF $\text{InvType}$
If it holds in the current state, and we perform a Next action, then it will hold in the next state.

Note that none of the Bug* definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

**THEOREM** ThmNextInvType

\[ \text{InvType} \land [\text{Next}]_{\text{vars}} \Rightarrow \text{InvType}' \]

**PROOF**

1. HAVE \( \text{InvType} \land [\text{Next}]_{\text{vars}} \)
2. USE DEF \text{InvType}
3. USE DEF \text{Pc}, \text{PcRecov}, \text{PcChkpt}
4. USE DEF \text{PcIDLE}
5. USE DEF \text{PcRECOV1}, \text{PcRECOV2}, \text{PcRECOV3}
6. USE DEF \text{PcCHKPT1}, \text{PcCHKPT2}, \text{PcCHKPT3}, \text{PcCHKPT4}, \text{PcCHKPT5}

Say QED here so that the rest of the proof has an indentation level. This creates a place where I can use the user interface renumbering operation to renumber all of the alternatives below.

1. QED
   Stutter step.
2. 1. CASE \( \text{vars}' = \text{vars} \)
   3. USE (2) 1
   3. USE DEF \text{vars}
   3. QED OBVIOUS

Walk through all Next alternatives.

2. 2. CASE NextObtainAccess
   3. USE NextObtainAccess
   3. USE DEF NextObtainAccess
   3. QED OBVIOUS

2. 3. CASE NextProveRevoke
   3. USE NextProveRevoke
   3. USE DEF NextProveRevoke
   3. \( \text{PcrPrior}(\text{appPcr}) \in \text{Pcr} \) BY ThmPcrPriorIsPcr
   3. QED BY ThmPcrExtendIsPcr

2. 4. CASE NextReboot
   3. USE NextReboot
   3. USE DEF NextReboot
   3. \( \text{InvType} ! \text{appPcr}' \) BY ThmAppRebootIsPcr
   3. \( \text{InvType} ! \text{semPcr}' \) BY ThmSemRebootIsPcr
   3. \( \text{InvType} ! \text{sealPcr}' \) BY ThmSealRebootIsPcr
   3. \( \text{InvType} ! \text{chkptts}' \) BY ThmNullTsIsTs
   3. QED OBVIOUS

2. 5. CASE NextForgetSealTs
   3. USE NextForgetSealTs
   3. USE DEF NextForgetSealTs
   3. QED OBVIOUS
(2)6. CASE NextExtendAppPcr
    (3) USE NextExtendAppPcr
    (3) USE DEF NextExtendAppPcr
    (3) appPcr' ∈ Pcr BY ThmPcrExtendIsPcr DEF Pcrx
    (3) QED OBVIOUS

(2)7. CASE NextExtendSemPcr
    (3) USE NextExtendSemPcr
    (3) USE DEF NextExtendSemPcr
    (3) InvType ! semPcr' BY ThmPcrExtendIsPcr DEF Pcrx
    (3) QED OBVIOUS

(2)8. CASE NextExtendSealPcr
    (3) USE NextExtendSealPcr
    (3) USE DEF NextExtendSealPcr
    (3) InvType ! sealPcr' BY ThmPcrExtendIsPcr DEF Pcrx
    (3) QED OBVIOUS

(2)9. CASE NextIncBootCtr
    (3) USE NextIncBootCtr
    (3) USE DEF NextIncBootCtr
    (3) bootCtr' ∈ Nat
        (4)1. bootCtr + 1 ∈ Nat BY SMT
        (4)2. bootCtr' = bootCtr + 1 OBVIOUS
    (4) QED BY ⟨4⟩1, ⟨4⟩2
    (3) QED OBVIOUS

(2)10. CASE NextEnterSemRecov
    (3) USE NextEnterSemRecov
    (3) USE DEF NextEnterSemRecov
    (3) InvType ! semPcr' BY ThmSemProtectIsPcr
    (3) InvType ! pc' OBVIOUS
    (3) InvType ! chkptts' OBVIOUS
    (3) QED OBVIOUS

(2)11. CASE NextSemRecov1 WhenCorrect
    (3) USE NextSemRecov1 WhenCorrect
    (3) USE DEF NextSemRecov1 WhenCorrect
    (3) QED OBVIOUS

(2)12. CASE NextSemRecov1 WhenIncorrect
    (3) USE NextSemRecov1 WhenIncorrect
    (3) USE DEF NextSemRecov1 WhenIncorrect
    (3) semPcr' ∈ Per BY ThmPcrExtendIsPer DEF Pcrx
    (3) QED OBVIOUS

(2)13. CASE NextSemRecov2
    (3) USE NextSemRecov2
\( 3 \) USE DEF NextSemRecover

\( 3 \) \( n\nu' \in N\nu \)

\( 4 \) 1. \( n\nu \in N\nu \) OBVIOUS

\( 4 \) 2. \( n\nu', current \in BOOLEAN \) BY DEF \( N\nu \)

\( 4 \) 3. \( n\nu' = [n\nu \text{ EXCEPT } !.current = n\nu'.current] \) OBVIOUS

\( 4 \) QED BY ONLY \( 4 \) 1, \( 4 \) 2, \( 4 \) 3 DEF \( N\nu \)

(3) QED OBVIOUS

(2) 14. CASE NextSemRecover

\( 3 \) USE NextSemRecover

\( 3 \) USE DEF NextSemRecover

\( 3 \) \( \text{semPcr}' \in \text{Pcr} \) BY ThmPerExtendIsPcr

\( 3 \) \( \text{pc}' \in \text{Pc} \) BY DEF \( \text{Pc}, \text{PcRecover} \)

(3) QED OBVIOUS

(2) 15. CASE NextSealTs

\( 3 \) USE NextSealTs

\( 3 \) USE DEF NextSealTs

\( 3 \) \( \text{sealPcr}' \in \text{Pcr} \) BY ThmPerExtendIsPcr

\( 3 \) \( \text{tsvalues}' \in \text{SUBSET} \text{Ts} \)

\( 4 \) 1. \( \text{tsvalues} \in \text{SUBSET} \text{Ts} \) OBVIOUS

\( 4 \) DEFINE \( ts := \text{NextSealTs}! : ts \)

\( 4 \) 2. \( ts \in \text{Ts} \) BY DEF \( \text{Ts, SignedTs} \)

\( 4 \) 3. \( \text{tsvalues}' = \text{tsvalues} \cup \{ts\} \) OBVIOUS

\( 4 \) QED BY \( 4 \) 1, \( 4 \) 2, \( 4 \) 3

(3) QED OBVIOUS

(2) 16. CASE NextEnterSemChkpt

\( 3 \) USE NextEnterSemChkpt

\( 3 \) USE DEF NextEnterSemChkpt

\( 3 \) \( \text{semPcr}' \in \text{Pcr} \) BY ThmSemProtectIsPcr

(3) QED OBVIOUS

(2) 17. CASE NextSemChkpt WhenCorrect

\( 3 \) USE NextSemChkpt WhenCorrect

\( 3 \) USE DEF NextSemChkpt WhenCorrect

\( 3 \) \( \text{chkpts}' \in \text{SignedTs} \)

\( 4 \) USE DEF EnterSemChkptPredicate

\( 4 \) USE DEF \( \text{Ts} \)

\( 4 \) QED OBVIOUS

(3) QED OBVIOUS

(2) 18. CASE NextSemChkpt WhenIncorrect

\( 3 \) USE NextSemChkpt WhenIncorrect

\( 3 \) USE DEF NextSemChkpt WhenIncorrect

\( 3 \) \( \text{semPcr}' \in \text{Pcr} \) BY ThmPerExtendIsPcr

(3) QED OBVIOUS
\( \langle 2 \rangle 19. \) **CASE** \( \text{NextSemChkpt2} \)

(3) **USE** \( \text{NextSemChkpt2} \)

(3) **USE DEF** \( \text{NextSemChkpt2} \)

(3) \( nv' \in Nv \)

(4) **DEFINE** \( \text{nvappPcr} 1 \triangleq \text{NextSemChkpt2}! : !\text{nvappPcr} 1 \)

(4) \( \text{nvappPcr} 1 \in \text{Pcr} \) **BY DEF** \( \text{SignedTs} \)

(4) **QED BY** \( \langle 4 \rangle 1 \) **DEF** \( Nv \)

(3) **QED OBVIOUS**

\( \langle 2 \rangle 20. \) **CASE** \( \text{NextSemChkpt3} \)

(3) **USE** \( \text{NextSemChkpt3} \)

(3) **USE DEF** \( \text{NextSemChkpt3} \)

(3) \( \text{bootCtr} ' \in \text{Nat} \)

(4) \( 1. \text{bootCtr} \in \text{Nat} \) **OBVIOUS**

(4) \( 2. \text{bootCtr} + 1 \in \text{Nat} \) **BY ONLY** \( \langle 4 \rangle 1, \text{SMT} \)

(4) **QED BY** \( \langle 4 \rangle 1, \langle 4 \rangle 2 \)

(3) **QED OBVIOUS**

\( \langle 2 \rangle 21. \) **CASE** \( \text{NextSemChkpt4} \)

(3) **USE** \( \text{NextSemChkpt4} \)

(3) **USE DEF** \( \text{NextSemChkpt4} \)

(3) \( nv' \in Nv \) **BY DEF** \( Nv \)

(3) **QED OBVIOUS**

\( \langle 2 \rangle 22. \) **CASE** \( \text{NextSemChkpt5} \)

(3) **USE** \( \text{NextSemChkpt5} \)

(3) **USE DEF** \( \text{NextSemChkpt5} \)

(3) \( \text{semPcr} ' \in \text{Pcr} \) **BY** \( \text{ThmPerExtendIsPcr} \) **DEF** \( \text{Pcrx} \)

(3) **QED OBVIOUS**

\( \langle 2 \rangle \) **QED**

**BY** \( \langle 2 \rangle 1, \)

\( \langle 2 \rangle 2, \langle 2 \rangle 3, \langle 2 \rangle 4, \langle 2 \rangle 5, \langle 2 \rangle 6, \langle 2 \rangle 7, \langle 2 \rangle 8, \langle 2 \rangle 9, \langle 2 \rangle 10, \)

\( \langle 2 \rangle 11, \langle 2 \rangle 12, \langle 2 \rangle 13, \langle 2 \rangle 14, \langle 2 \rangle 15, \langle 2 \rangle 16, \langle 2 \rangle 17, \langle 2 \rangle 18, \)

\( \langle 2 \rangle 19, \langle 2 \rangle 20, \langle 2 \rangle 21, \langle 2 \rangle 22 \)

**DEF** \( \text{Next} \)

---

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

\( \text{Spec} \Rightarrow \Box \text{InvType} \)

**PROOF**

(1) \( \text{Init} \Rightarrow \text{InvType} \) **BY** \( \text{ThmInitInvType} \)

(1) \( \text{InvType} \land [\text{Next}]_{\text{vars}} \Rightarrow \text{InvType'} \) **BY** \( \text{ThmNextInvType} \)

(1) **QED**
PROOF OF INVARIANT InvInSemProtect

It holds in the initial state.

THEOREM ThmInitInvInSemProtect \[ \Delta \]
\[ \text{Init} \Rightarrow \text{InvInSemProtect} \]
PROOF
\begin{enumerate}
\item HAVE \text{Init}
\item \text{InvType} BY ThmInitInvType
\item USE DEF \text{Init}
\item USE DEF \text{InSem}
\item QED BY DEF InvInSemProtect
\end{enumerate}

If it holds in the current state, and we perform a \text{Next} action, then it will hold in the next state.

Note that none of the Bug* definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

THEOREM ThmNextInvInSemProtect \[ \Delta \]
\[ \text{InvInSemProtect} \land [\text{Next}]_{\text{vars}} \Rightarrow \text{InvInSemProtect}' \]
PROOF
\begin{enumerate}
\item HAVE \text{InvInSemProtect} \land [\text{Next}]_{\text{vars}}
\item USE DEF \text{InvInSemProtect}
\item USE DEF \text{InSem}
\item \text{InvType}' BY ThmNextInvType
\item \text{InvInSemProtect}' \text{!goal}'
\item USE DEF \text{PcIDLE}
\item USE DEF \text{PcRECOV1}, \text{PcRECOV2}, \text{PcRECOV3}
\item USE DEF \text{PcCHKPT1}, \text{PcCHKPT2}, \text{PcCHKPT3}, \text{PcCHKPT4}, \text{PcCHKPT5}
\end{enumerate}

Stutter step.
\begin{enumerate}
\item CASE \text{vars}' = \text{vars}
\item USE (2)1
\item USE DEF \text{vars}
\item QED OBVIOUS
\end{enumerate}

Walk through all \text{Next} alternatives.
\begin{enumerate}
\item CASE \text{NextObtainAccess}
\item USE \text{NextObtainAccess}
\item USE DEF \text{NextObtainAccess}
\item QED OBVIOUS
\item CASE \text{NextProveRevoke}
\item USE \text{NextProveRevoke}
\end{enumerate}
\text{C \ PROOF OF INITIAL PASTURE}
\[2\] 12. CASE \textit{NextSemRecov}1 \textit{WhenIncorrect}  
(3) USE \textit{NextSemRecov}1 \textit{WhenIncorrect}  
(3) USE DEF \textit{NextSemRecov}1 \textit{WhenIncorrect}  
(3) USE DEF \textit{PcrExtend}  
(3) QED OBVIOUS  
\[2\] 13. CASE \textit{NextSemRecov}2  
(3) USE \textit{NextSemRecov}2  
(3) USE DEF \textit{NextSemRecov}2  
(3) QED OBVIOUS  
\[2\] 14. CASE \textit{NextSemRecov}3  
(3) USE \textit{NextSemRecov}3  
(3) USE DEF \textit{NextSemRecov}3  
(3) USE DEF \textit{PcrExtend}  
(3) QED OBVIOUS  
\[2\] 15. CASE \textit{NextSealTs}  
(3) USE \textit{NextSealTs}  
(3) USE DEF \textit{NextSealTs}  
(3) QED OBVIOUS  
\[2\] 16. CASE \textit{NextEnterSemChkpt}  
(3) USE \textit{NextEnterSemChkpt}  
(3) USE DEF \textit{NextEnterSemChkpt}  
(3) QED OBVIOUS  
\[2\] 17. CASE \textit{NextSemChkpt}1 \textit{WhenCorrect}  
(3) USE \textit{NextSemChkpt}1 \textit{WhenCorrect}  
(3) USE DEF \textit{NextSemChkpt}1 \textit{WhenCorrect}  
(3) QED OBVIOUS  
\[2\] 18. CASE \textit{NextSemChkpt}1 \textit{WhenIncorrect}  
(3) USE \textit{NextSemChkpt}1 \textit{WhenIncorrect}  
(3) USE DEF \textit{NextSemChkpt}1 \textit{WhenIncorrect}  
(3) USE DEF \textit{PcrExtend}  
(3) QED OBVIOUS  
\[2\] 19. CASE \textit{NextSemChkpt}2  
(3) USE \textit{NextSemChkpt}2  
(3) USE DEF \textit{NextSemChkpt}2  
(3) QED OBVIOUS  
\[2\] 20. CASE \textit{NextSemChkpt}3  
(3) USE \textit{NextSemChkpt}3  
(3) USE DEF \textit{NextSemChkpt}3  
(3) QED OBVIOUS  
\[2\] 21. CASE \textit{NextSemChkpt}4
\[ \langle 3 \rangle \text{ USE } \text{NextSemChkpt4} \]
\[ \langle 3 \rangle \text{ USE DEF } \text{NextSemChkpt4} \]
\[ \langle 3 \rangle \text{ QED OBVIOUS} \]

\[ \langle 2 \rangle \text{22. CASE } \text{NextSemChkpt5} \]
\[ \langle 3 \rangle \text{ USE } \text{NextSemChkpt5} \]
\[ \langle 3 \rangle \text{ USE DEF } \text{NextSemChkpt5} \]
\[ \langle 3 \rangle \text{ USE DEF } \text{PcrExtend} \]
\[ \langle 3 \rangle \text{ QED OBVIOUS} \]

\[ \langle 2 \rangle \text{ QED} \]
\[ \text{BY } \langle 2 \rangle 1, \]
\[ \langle 2 \rangle 2, \langle 2 \rangle 3, \langle 2 \rangle 4, \langle 2 \rangle 5, \langle 2 \rangle 6, \langle 2 \rangle 7, \langle 2 \rangle 8, \langle 2 \rangle 9, \langle 2 \rangle 10, \]
\[ \langle 2 \rangle 11, \langle 2 \rangle 12, \langle 2 \rangle 13, \langle 2 \rangle 14, \langle 2 \rangle 15, \langle 2 \rangle 16, \langle 2 \rangle 17, \langle 2 \rangle 18, \]
\[ \langle 2 \rangle 19, \langle 2 \rangle 20, \langle 2 \rangle 21, \langle 2 \rangle 22 \]
\[ \text{DEF } \text{Next} \]
\[ \langle 1 \rangle \text{ QED OBVIOUS} \]

It is an invariant of the specification.

**THEOREM** \( \text{ThmInvInSemProtect} \triangleq \)
\[ \text{Spec} \Rightarrow \Box \text{InvInSemProtect} \]

**PROOF**
\[ \langle 1 \rangle \text{ Init } \Rightarrow \text{InvInSemProtect BY ThmInitInvInSemProtect} \]
\[ \langle 1 \rangle \text{ InvInSemProtect } \land \lbrack \text{Next} \rbrack_{\text{vars}} \Rightarrow \text{InvInSemProtect}' \]
\[ \text{BY ThmNextInvInSemProtect} \]
\[ \langle 1 \rangle \text{ USE DEF Spec} \]
\[ \langle 1 \rangle \text{ QED} \]

---

**PROOF OF INVARIANT InvUnreachableSemProtect**

It holds in the initial state.

**THEOREM** \( \text{ThmInitInvUnreachableSemProtect} \triangleq \)
\[ \text{Init } \Rightarrow \text{InvUnreachableSemProtect} \]

**PROOF**
\[ \langle 1 \rangle \text{ HAVE Init} \]
\[ \langle 1 \rangle \text{ InvType BY ThmInitInvType} \]
(1) \textit{InvInSemProtect} \textbf{BY} \textit{ThmInitInvInSemProtect}
(1) USE DEF Init
(1) USE DEF InSem
(1) \neg InSem \textbf{OBVIOUS}
(1) \neg PerLeq(semPcr, SemProtect)
(2) USE DEF PerLeq
(2) USE DEF SemProtect
(2) USE DEF SemReboot
(2) USE DEF Pcri
(2) USE DEF PcrInit
(2) USE DEF PcrLeq
(2) \textbf{QED BY} \textit{AssSemProtect}
(1) \textbf{QED BY} DEF \textit{InvUnreachableSemProtect}

If it holds in the current state, and we perform a \textit{Next} action, then it will hold in the next state.

Note that none of the Bug* definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

\textbf{THEOREM} \textit{ThmNextInvUnreachableSemProtect} \iff \textit{InvUnreachableSemProtect} \land \lbrack Next \rbrack \textit{vars} \Rightarrow \textit{InvUnreachableSemProtect'}

\textbf{PROOF}
(1) \textbf{HAVE} \textit{InvUnreachableSemProtect} \land \lbrack Next \rbrack \textit{vars}
(1) USE DEF \textit{InvUnreachableSemProtect}
(1) USE DEF InSem
(1) USE DEF \textit{InvInSemProtect}
(1) \textit{InvInSemProtect'} \textbf{BY} \textit{ThmNextInvType}
(1) \textit{InvInSemProtect'} \textbf{BY} \textit{ThmNextInvInSemProtect}
(1) \textit{InvUnreachableSemProtect'} \textbf{!goal'}
(2) USE DEF PcIDLE
(2) USE DEF PcRECOV1, PcRECOV2, PcRECOV3
(2) USE DEF PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5

\textbf{Stutter step.}
(2) \textbf{1. CASE} \textit{vars'} = \textit{vars}
(3) USE (2) 1
(3) USE DEF \textit{vars}
(3) \textbf{QED OBVIOUS}

\textbf{Walk through all \textit{Next} alternatives.}
(2) \textbf{2. CASE} \textit{NextObtainAccess}
(3) USE \textit{NextObtainAccess}
(3) USE DEF \textit{NextObtainAccess}
(3) \textbf{QED OBVIOUS}
(2) \textbf{3. CASE} \textit{NextProveRevoke}
(3) USE NextProveRevoke
(3) USE DEF NextProveRevoke
(3) QED OBVIOUS

(2)4. CASE NextReboot
(3) USE NextReboot
(3) USE DEF NextReboot
(3) USE DEF PcrLeq
(3) USE DEF SemProtect
(3) USE DEF SemReboot
(3) USE DEF Pcri
(3) USE DEF AssSemProtect
(3) QED OBVIOUS

(2)5. CASE NextForgetSealTs
(3) USE NextForgetSealTs
(3) USE DEF NextForgetSealTs
(3) QED OBVIOUS

(2)6. CASE NextExtendAppPcr
(3) USE NextExtendAppPcr
(3) USE DEF NextExtendAppPcr
(3) QED OBVIOUS

(2)7. CASE NextExtendSemPcr
(3) USE NextExtendSemPcr
(3) USE DEF NextExtendSemPcr
(3) USE DEF InvType
(3) ¬PcrLeq(semPcr, SemProtect)
   (4) SemProtect ∈ Pcr BY ThmSemProtectIsPcr
   (4) QED BY ThmPcrExtendSourceUnreachable
(3) QED OBVIOUS

(2)8. CASE NextExtendSealPcr
(3) USE NextExtendSealPcr
(3) USE DEF NextExtendSealPcr
(3) QED OBVIOUS

(2)9. CASE NextIncBootCtr
(3) USE NextIncBootCtr
(3) USE DEF NextIncBootCtr
(3) QED OBVIOUS

(2)10. CASE NextEnterSemRecov
(3) USE NextEnterSemRecov
(3) USE DEF NextEnterSemRecov
(3) QED OBVIOUS
11. CASE NextSemRecov1 WhenCorrect
   (3) USE NextSemRecov1 WhenCorrect
   (3) USE DEF NextSemRecov1 WhenCorrect
   (3) QED OBVIOUS

12. CASE NextSemRecov1 WhenIncorrect
   (3) USE NextSemRecov1 WhenIncorrect
   (3) USE DEF NextSemRecov1 WhenIncorrect
   (3) USE DEF InvType
   (3) USE DEF Pcrx
   (3) \( \neg \text{PcrLeq}(\text{semPcr}', \text{SemProtect}) \)
   (4) QED BY ThmPcrExtendSelfUnreachable
   (3) QED OBVIOUS

13. CASE NextSemRecov2
   (3) USE NextSemRecov2
   (3) USE DEF NextSemRecov2
   (3) QED OBVIOUS

14. CASE NextSemRecov3
   (3) USE NextSemRecov3
   (3) USE DEF NextSemRecov3
   (3) USE DEF InvType
   (3) USE DEF Pcrx
   (3) \( \neg \text{PcrLeq}(\text{semPcr}', \text{SemProtect}) \)
   (4) QED BY ThmPcrExtendSelfUnreachable
   (3) QED OBVIOUS

15. CASE NextSealTs
   (3) USE NextSealTs
   (3) USE DEF NextSealTs
   (3) QED OBVIOUS

16. CASE NextEnterSemChkpt
   (3) USE NextEnterSemChkpt
   (3) USE DEF NextEnterSemChkpt
   (3) QED OBVIOUS

17. CASE NextSemChkpt1 WhenCorrect
   (3) USE NextSemChkpt1 WhenCorrect
   (3) USE DEF NextSemChkpt1 WhenCorrect
   (3) QED OBVIOUS

18. CASE NextSemChkpt1 WhenIncorrect
   (3) USE NextSemChkpt1 WhenIncorrect
   (3) USE DEF NextSemChkpt1 WhenIncorrect
   (3) USE DEF InvType
   (3) USE DEF Pcrx
It is an invariant of the specification.

**Theorem** ThmInvUnreachableSemProtect \( \equiv \)
\[
\text{Spec} \Rightarrow \Box \text{InvUnreachableSemProtect}
\]

**Proof**

1. \( \text{Init} \Rightarrow \text{InvUnreachableSemProtect} \) by ThmInitInvUnreachableSemProtect
2. \( \text{InvUnreachableSemProtect} \land [\text{Next}]_{\text{vars}} \Rightarrow \text{InvUnreachableSemProtect}' \) by ThmNextInvUnreachableSemProtect
PROOF OF INVARIANT InvNuProtection

It is an invariant of the specification.

THEOREM ThmInvNuProtection ≜
Spec ⇒ □ InvNuProtection

PROOF
(1) InvInSemProtect ∧ InvUnreachableSemProtect ⇒ InvNuProtection
(2) HAVE InvInSemProtect ∧ InvUnreachableSemProtect
(2) USE DEF InvInSemProtect
(2) USE DEF InvUnreachableSemProtect
(2) USE DEF InvNuProtection

(2) 1. CASE InSem
(2) BY (2) 1

(2) 2. CASE ¬ InSem
Proof by contradiction.
(3) 1. CASE semPcr ≠ SemProtect
(3) BY (3) 1
(3) 2. CASE semPcr = SemProtect
(4) 1. ¬PcrLeq(semPcr, SemProtect)
(4) BY (2) 2
(4) 2. PcrLeq(semPcr, SemProtect)
(4) QED BY (3) 2, ThmPcrLeqIsReflexive
(4) QED BY (4) 1, (4) 2
(3) QED BY (3) 2, (3) 1
(2) QED BY (2) 2, (2) 1

(1) Spec ⇒ □ InvInSemProtect
(1) Spec ⇒ □ InvUnreachableSemProtect
(1) QED
It holds in the initial state.

**THEOREM** \( \text{ThmInitInvSignedTsLeqBoot} \triangleq \text{Init} \Rightarrow \text{InvSignedTsLeqBoot} \)

**PROOF**

1. **HAVE** \( \text{Init} \)
2. **USE DEF** \( \text{Init} \)
3. **USE DEF** \( \text{InvSignedTsLeqBoot} \)
   - **InvSignedTsLeqBoot** \( \downarrow \) goal
4. **TAKE** \( ts \in \text{tsvalues} \cup \{\text{chkptts}\} \)
5. **QED BY** \( \text{ThmNullTsIsntSignedTs} \)
6. **QED OBVIOUS**

If it holds in the current state, and we perform a \( \text{Next} \) action, then it will hold in the next state.

Note that none of the Bug* definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

**THEOREM** \( \text{ThmNextInvSignedTsLeqBoot} \triangleq \text{InvSignedTsLeqBoot} \land [\text{Next}]_{\text{vars}} \Rightarrow \text{InvSignedTsLeqBoot}' \)

**PROOF**

1. **HAVE** \( \text{InvSignedTsLeqBoot} \land [\text{Next}]_{\text{vars}} \)
2. **USE DEF** \( \text{InvSignedTsLeqBoot} \)
3. **InvSignedTsLeqBoot** \( \downarrow \) goal'

   **Stutter step.**

   1. **CASE** \( \text{vars}' = \text{vars} \)
      - **USE** \( (2)1 \)
      - **USE DEF** \( \text{vars} \)
      - **QED OBVIOUS**

   **Walk through all \( \text{Next} \) alternatives.**

   2. **CASE** \( \text{NextObtainAccess} \)
      - **USE** \( \text{NextObtainAccess} \)
      - **USE DEF** \( \text{NextObtainAccess} \)
      - **QED OBVIOUS**

   3. **CASE** \( \text{NextProveRevoke} \)
      - **USE** \( \text{NextProveRevoke} \)
      - **USE DEF** \( \text{NextProveRevoke} \)
      - **QED OBVIOUS**

   4. **CASE** \( \text{NextReboot} \)
      - **USE** \( \text{NextReboot} \)
      - **USE DEF** \( \text{NextReboot} \)
(3) QED BY ThmNullTsIsnSignedTs

(2) 5. CASE NextForgetSealTs
   (3) USE NextForgetSealTs
   (3) USE DEF NextForgetSealTs
   (3) QED OBVIOUS

(2) 6. CASE NextExtendAppPcr
   (3) USE NextExtendAppPcr
   (3) USE DEF NextExtendAppPcr
   (3) QED OBVIOUS

(2) 7. CASE NextExtendSemPcr
   (3) USE NextExtendSemPcr
   (3) USE DEF NextExtendSemPcr
   (3) QED OBVIOUS

(2) 8. CASE NextExtendSealPcr
   (3) USE NextExtendSealPcr
   (3) USE DEF NextExtendSealPcr
   (3) QED OBVIOUS

(2) 9. CASE NextIncBootCtr
   (3) USE NextIncBootCtr
   (3) USE DEF NextIncBootCtr
   (3) USE DEF InvType
   (3) USE DEF SignedTs
   (3) bootCtr ≤ bootCtr + 1 BY ThmNatMore
   (3) QED BY ThmNatLeqIsTransitive

(2) 10. CASE NextEnterSemRecov
   (3) USE NextEnterSemRecov
   (3) USE DEF NextEnterSemRecov
   (3) QED OBVIOUS

(2) 11. CASE NextSemRecov1 WhenCorrect
   (3) USE NextSemRecov1 WhenCorrect
   (3) USE DEF NextSemRecov1 WhenCorrect
   (3) QED OBVIOUS

(2) 12. CASE NextSemRecov1 WhenIncorrect
   (3) USE NextSemRecov1 WhenIncorrect
   (3) USE DEF NextSemRecov1 WhenIncorrect
   (3) QED OBVIOUS

(2) 13. CASE NextSemRecov2
   (3) USE NextSemRecov2
   (3) USE DEF NextSemRecov2
   (3) QED OBVIOUS
(2.14. CASE NextSemRecov3
(3) USE NextSemRecov3
(3) USE DEF NextSemRecov3
(3) QED OBVIOUS

(2.15. CASE NextSealTs
(3) USE NextSealTs
(3) USE DEF NextSealTs
(3) DEFINE ts = NextSealTs : !ts
(3) ts.bootCtr ≤ bootCtr
(4) USE DEF InvType
(4) USE ThmNatLeqIsReflexive
(4) QED OBVIOUS
(3) QED OBVIOUS

(2.16. CASE NextEnterSemChkpt
(3) USE NextEnterSemChkpt
(3) USE DEF NextEnterSemChkpt
(3) QED OBVIOUS

(2.17. CASE NextSemChkpt1 WhenCorrect
(3) USE NextSemChkpt1 WhenCorrect
(3) USE DEF NextSemChkpt1 WhenCorrect
(3) QED OBVIOUS

(2.18. CASE NextSemChkpt1 WhenIncorrect
(3) USE NextSemChkpt1 WhenIncorrect
(3) USE DEF NextSemChkpt1 WhenIncorrect
(3) QED OBVIOUS

(2.19. CASE NextSemChkpt2
(3) USE NextSemChkpt2
(3) USE DEF NextSemChkpt2
(3) QED OBVIOUS

(2.20. CASE NextSemChkpt3
(3) USE NextSemChkpt3
(3) USE DEF NextSemChkpt3
(3) USE DEF InvType
(3) USE DEF SignedTs
(3) bootCtr ≤ bootCtr'
(4) USE ThmNatLeqIsReflexive
(4) USE ThmNatMore
(4) QED OBVIOUS
(3) USE ThmNatLeqIsTransitive
(3) QED OBVIOUS

(2.21. CASE NextSemChkpt4
PROOF OF INVARIANT \textit{InvUnforgeableSemHappy}

It holds in the initial state.

\textbf{THEOREM} \textit{ThmInitInvUnforgeableSemHappy} \ \Rightarrow \ \textit{Init} \Rightarrow \textit{InvUnforgeableSemHappy}

\textbf{PROOF}

(1) \textbf{HAVE} \textit{Init}
(1) \textbf{InvType} \textbf{BY} \textit{ThmInitInvType}
(1) \textbf{InvInSemProtect} \textbf{BY} \textit{ThmInitInvInSemProtect}
(1) \textbf{USE DEF Init}
(1) \textbf{USE DEF InSem}
(1) \neg \textit{InSemOBVIOUS}
(1) \neg \textit{PcrLeq(semPcr, SemHappy)}
(2) \textbf{USE DEF PcrLeq}
(2) \textbf{USE DEF PcrInit}
(2) \textbf{USE DEF PcrExtend}
(2) \textbf{USE DEF SemHappy}
(2) \textbf{USE DEF SemReboot}
(2) \textbf{USE DEF SemProtect}
(2) \textbf{USE AssSemProtect}

\begin{itemize}
  \item \langle 3 \rangle \textbf{USE NextSemChkpt4}
  \item \langle 3 \rangle \textbf{USE DEF NextSemChkpt4}
  \item \langle 3 \rangle \textbf{QED OBVIOUS}
\end{itemize}

\begin{itemize}
  \item \langle 2 \rangle \textbf{22. CASE NextSemChkpt5}
  \item \langle 3 \rangle \textbf{USE NextSemChkpt5}
  \item \langle 3 \rangle \textbf{USE DEF NextSemChkpt5}
  \item \langle 3 \rangle \textbf{QED OBVIOUS}
\end{itemize}

\begin{itemize}
  \item \langle 2 \rangle \textbf{QED}
  \item \langle 2 \rangle \textbf{BY} \langle 2 \rangle \textbf{1}, \langle 2 \rangle \textbf{2}, \langle 2 \rangle \textbf{3}, \langle 2 \rangle \textbf{4}, \langle 2 \rangle \textbf{5}, \langle 2 \rangle \textbf{6}, \langle 2 \rangle \textbf{7}, \langle 2 \rangle \textbf{8}, \langle 2 \rangle \textbf{9}, \langle 2 \rangle \textbf{10}, \langle 2 \rangle \textbf{11}, \langle 2 \rangle \textbf{12}, \langle 2 \rangle \textbf{13}, \langle 2 \rangle \textbf{14}, \langle 2 \rangle \textbf{15}, \langle 2 \rangle \textbf{16}, \langle 2 \rangle \textbf{17}, \langle 2 \rangle \textbf{18}, \langle 2 \rangle \textbf{19}, \langle 2 \rangle \textbf{20}, \langle 2 \rangle \textbf{21}, \langle 2 \rangle \textbf{22}
  \item \textbf{DEF Next}
  \item \langle 1 \rangle \textbf{QED OBVIOUS}
\end{itemize}
If it holds in the current state, and we perform a Next action, then it will hold in the next state.

Note that none of the Bug* definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

**THEOREM** ThmNextInvUnforgeableSemHappy

\[ \text{InvUnforgeableSemHappy} \land [\text{Next}]_{\text{vars}} \Rightarrow \text{InvUnforgeableSemHappy}' \]

**PROOF**

1. **HAVE** InvUnforgeableSemHappy \& [Next]_{vars}
2. **USE** DEF InvUnforgeableSemHappy

1. **USE** DEF InSem

1. **InvType** \& BY ThmNextInvType
2. **USE** InvInSemProtect \& BY ThmNextInvInSemProtect
3. **USE** InvUnforgeableSemHappy \& goal'

1. **USE** DEF PcIDLE
2. **USE** DEF PcRECOV1, PcRECOV2, PcRECOV3
3. **USE** DEF PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5

**Stutter step.**

1. **CASE** var = var'
2. **USE** (2) 1
3. **USE** DEF var
4. **QED OBVIOUS**

**Walk through all Next alternatives.**

1. **CASE** NextObtainAccess
2. **USE** NextObtainAccess
3. **USE** DEF NextObtainAccess
4. **QED OBVIOUS**

1. **CASE** NextProveRevoke
2. **USE** NextProveRevoke
3. **USE** DEF NextProveRevoke
4. **QED OBVIOUS**

1. **CASE** NextReboot
2. **USE** NextReboot
3. **USE** DEF NextReboot
4. **USE** DEF PcLeq
5. **USE** DEF PcrInit
6. **USE** DEF PcrExtend
7. **USE** DEF SemHappy
8. **USE** DEF SemReboot
(3) USE DEF SemProtect
(3) USE AssSemProtect
(3) QED OBVIOUS

(2) 5. CASE NextForgetSealTs
(3) USE NextForgetSealTs
(3) USE DEF NextForgetSealTs
(3) QED OBVIOUS

(2) 6. CASE NextExtendAppPcr
(3) USE NextExtendAppPcr
(3) USE DEF NextExtendAppPcr
(3) QED OBVIOUS

(2) 7. CASE NextExtendSemPcr
(3) USE NextExtendSemPcr
(3) USE DEF NextExtendSemPcr
(3) USE DEF InvType
(3) HAVE ¬InSem′
(3) 1. CASE semPcr = SemHappy
   (4) USE (3) 1
   (4) QED BY ThmPcrExtendSelfUnreachable
(3) 2. CASE ¬PcrLeq(semPcr, SemHappy)
   (4) USE (3) 2
   (4) SemHappy ∈ Pcr BY ThmSemHappyIsPcr
   (4) ¬PcrLeq(semPcr′, SemHappy) BY ThmPcrExtendSourceUnreachable
   (4) QED OBVIOUS
(3) QED BY (3) 1, (3) 2

(2) 8. CASE NextExtendSealPcr
(3) USE NextExtendSealPcr
(3) USE DEF NextExtendSealPcr
(3) QED OBVIOUS

(2) 9. CASE NextIncBootCtr
(3) USE NextIncBootCtr
(3) USE DEF NextIncBootCtr
(3) QED OBVIOUS

(2) 10. CASE NextEnterSemRecov
(3) USE NextEnterSemRecov
(3) USE DEF NextEnterSemRecov
(3) QED OBVIOUS

(2) 11. CASE NextSemRecov1WhenCorrect
(3) USE NextSemRecov1WhenCorrect
(3) USE DEF NextSemRecov1WhenCorrect
(3) QED OBVIOUS
\(2\). 12. CASE \textit{NextSemRecov1}\textit{WhenIncorrect}  
(3) USE \textit{NextSemRecov1}\textit{WhenIncorrect}  
(3) USE DEF \textit{NextSemRecov1}\textit{WhenIncorrect}  
(3) USE DEF InvType  
(3) USE DEF \textit{Pcrx}  
(3) \(\text{semPcr} = \text{SemProtect}\) BY DEF InvInSemProtect  
(3) USE DEF SemHappy  
(3) USE AssSemHappy  
(3) USE ThmPcrExtendIncompatible  
(3) QED OBVIOUS

\(2\). 13. CASE \textit{NextSemRecov2}  
(3) USE \textit{NextSemRecov2}  
(3) USE DEF \textit{NextSemRecov2}  
(3) QED OBVIOUS

\(2\). 14. CASE \textit{NextSemRecov3}  
(3) USE \textit{NextSemRecov3}  
(3) USE DEF \textit{NextSemRecov3}  
(3) \(\text{semPcr} = \text{SemProtect}\) BY DEF InvInSemProtect  
(3) USE DEF SemHappy  
(3) QED OBVIOUS

\(2\). 15. CASE \textit{NextSealTs}  
(3) USE \textit{NextSealTs}  
(3) USE DEF \textit{NextSealTs}  
(3) QED OBVIOUS

\(2\). 16. CASE \textit{NextEnterSemChkpt}  
(3) USE \textit{NextEnterSemChkpt}  
(3) USE DEF \textit{NextEnterSemChkpt}  
(3) QED OBVIOUS

\(2\). 17. CASE \textit{NextSemChkpt1}\textit{WhenCorrect}  
(3) USE \textit{NextSemChkpt1}\textit{WhenCorrect}  
(3) USE DEF \textit{NextSemChkpt1}\textit{WhenCorrect}  
(3) QED OBVIOUS

\(2\). 18. CASE \textit{NextSemChkpt1}\textit{WhenIncorrect}  
(3) USE \textit{NextSemChkpt1}\textit{WhenIncorrect}  
(3) USE DEF \textit{NextSemChkpt1}\textit{WhenIncorrect}  
(3) USE DEF InvType  
(3) USE DEF \textit{Pcrx}  
(3) \(\text{semPcr} = \text{SemProtect}\) BY DEF InvInSemProtect  
(3) USE DEF SemHappy  
(3) USE AssSemHappy  
(3) USE ThmPcrExtendIncompatible  
(3) QED OBVIOUS
\(\langle 2 \rangle 19. \text{CASE NextSemChkpt2} \)
\(\langle 3 \rangle \text{USE NextSemChkpt2} \)
\(\langle 3 \rangle \text{USE DEF NextSemChkpt2} \)
\(\langle 3 \rangle \text{QED OBVIOUS} \)

\(\langle 2 \rangle 20. \text{CASE NextSemChkpt3} \)
\(\langle 3 \rangle \text{USE NextSemChkpt3} \)
\(\langle 3 \rangle \text{USE DEF NextSemChkpt3} \)
\(\langle 3 \rangle \text{QED OBVIOUS} \)

\(\langle 2 \rangle 21. \text{CASE NextSemChkpt4} \)
\(\langle 3 \rangle \text{USE NextSemChkpt4} \)
\(\langle 3 \rangle \text{USE DEF NextSemChkpt4} \)
\(\langle 3 \rangle \text{QED OBVIOUS} \)

\(\langle 2 \rangle 22. \text{CASE NextSemChkpt5} \)
\(\langle 3 \rangle \text{USE NextSemChkpt5} \)
\(\langle 3 \rangle \text{USE DEF NextSemChkpt5} \)
\(\langle 3 \rangle \text{USE DEF InvType} \)
\(\langle 3 \rangle \text{USE DEF Pcrx} \)
\(\langle 3 \rangle \text{semPcr = SemProtect} \text{BY DEF InvInSemProtect} \)
\(\langle 3 \rangle \text{USE DEF SemHappy} \)
\(\langle 3 \rangle \text{USE AssSemHappy} \)
\(\langle 3 \rangle \text{USE ThmPcrExtendIncompatible} \)
\(\langle 3 \rangle \text{QED OBVIOUS} \)

\(\langle 2 \rangle \text{QED} \)
\(\text{BY (2)1,} \)
\(\langle 2 \rangle 2, \langle 2 \rangle 3, \langle 2 \rangle 4, \langle 2 \rangle 5, \langle 2 \rangle 6, \langle 2 \rangle 7, \langle 2 \rangle 8, \langle 2 \rangle 9, \langle 2 \rangle 10, \)
\(\langle 2 \rangle 11, \langle 2 \rangle 12, \langle 2 \rangle 13, \langle 2 \rangle 14, \langle 2 \rangle 15, \langle 2 \rangle 16, \langle 2 \rangle 17, \langle 2 \rangle 18, \)
\(\langle 2 \rangle 19, \langle 2 \rangle 20, \langle 2 \rangle 21, \langle 2 \rangle 22 \)
\(\text{DEF Next} \)
\(\langle 1 \rangle \text{QED OBVIOUS} \)

\text{It is an invariant of the specification.}

\text{THEOREM ThmInvUnforgeableSemHappy} \triangleq \text{Spec} \Rightarrow \square \text{InvUnforgeableSemHappy} \)
\text{PROOF} \)
\(\langle 1 \rangle \text{Init} \Rightarrow \text{InvUnforgeableSemHappy} \text{BY ThmInitInvUnforgeableSemHappy} \)
\(\langle 1 \rangle \text{InvUnforgeableSemHappy} \land \text{[Next]} \text{vars} \Rightarrow \text{InvUnforgeableSemHappy'} \text{BY ThmNextInvUnforgeableSemHappy} \)
\(\langle 1 \rangle \text{USE DEF Spec} \)
\(\langle 1 \rangle \text{QED} \)
PROOF OF INVARIANT $InvUnforgeableSealReboot$

It holds in the initial state.

**THEOREM** $ThmInitInvUnforgeableSealReboot \triangleq$

$Init \Rightarrow InvUnforgeableSealReboot$

**PROOF**

1. **HAVE** $Init$
2. **InvType** BY $ThmInitInvType$
3. $sealPcr = SealReboot$ BY DEF $Init$
4. **QED** BY DEF $InvUnforgeableSealReboot$

If it holds in the current state, and we perform a $Next$ action, then it will hold in the next state.

Note that none of the Bug* definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

**THEOREM** $ThmNextInvUnforgeableSealReboot \triangleq$

$InvUnforgeableSealReboot \land [Next]_{vars} \Rightarrow InvUnforgeableSealReboot'$

**PROOF**

1. **HAVE** $InvUnforgeableSealReboot \land [Next]_{vars}$
2. **USE** DEF $InvUnforgeableSealReboot$
3. **InvType'$ BY $ThmNextInvType$
4. **InvUnforgeableSealReboot ! goal'$

   1. **USE** DEF $PcIDLE$
   2. **USE** DEF $PcRECOV1, PcRECOV2, PcRECOV3$
   3. **USE** DEF $PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5$

Stutter step.

1. **CASE** $vars' = vars$
   1. **USE** (2) 1
   2. **USE** DEF $vars$
   3. **QED OBVIOUS$

Walk through all $Next$ alternatives.

1. **CASE** $NextObtainAccess$
   1. **USE** $NextObtainAccess$
   2. **USE** DEF $NextObtainAccess$
   3. **QED OBVIOUS$

2. **CASE** $NextProveRevoke$
   1. **USE** $NextProveRevoke$
   2. **USE** DEF $NextProveRevoke$
   3. **QED OBVIOUS
(2) 4. CASE NextReboot
    (3) USE NextReboot
    (3) USE DEF NextReboot
    (3) QED OBVIOUS

(2) 5. CASE NextForgetSealTs
    (3) USE NextForgetSealTs
    (3) USE DEF NextForgetSealTs
    (3) QED OBVIOUS

(2) 6. CASE NextExtendAppPcr
    (3) USE NextExtendAppPcr
    (3) USE DEF NextExtendAppPcr
    (3) QED OBVIOUS

(2) 7. CASE NextExtendSemPcr
    (3) USE NextExtendSemPcr
    (3) USE DEF NextExtendSemPcr
    (3) QED OBVIOUS

(2) 8. CASE NextExtendSealPcr
    (3) USE NextExtendSealPcr
    (3) USE DEF NextExtendSealPcr
    (3) sealPcr ∈ Pcr
    (3) SealReboot ∈ Pcr
    (3) QED BY ThmSealRebootIsPcr

(2) 9. CASE NextIncBootCtr
    (3) USE NextIncBootCtr
    (3) USE DEF NextIncBootCtr
    (3) QED OBVIOUS

(2) 10. CASE NextEnterSemRecov
    (3) USE NextEnterSemRecov
    (3) USE DEF NextEnterSemRecov
    (3) QED OBVIOUS

(2) 11. CASE NextSemRecov1 WhenCorrect
    (3) USE NextSemRecov1 WhenCorrect
    (3) USE DEF NextSemRecov1 WhenCorrect
    (3) QED OBVIOUS

(2) 12. CASE NextSemRecov1 WhenIncorrect
    (3) USE NextSemRecov1 WhenIncorrect
    (3) USE DEF NextSemRecov1 WhenIncorrect
    (3) QED OBVIOUS

(2) 13. CASE NextSemRecov2
    (3) USE NextSemRecov2
\(\text{(3) USE DEF } \textit{NextSemRecov}^2\)

\(\text{(3) QED OBVIOUS}\)

(2.14. CASE \textit{NextSemRecov}^3)

\(\text{(3) USE } \textit{NextSemRecov}^3\)

\(\text{(3) USE DEF } \textit{NextSemRecov}^3\)

\(\text{(3) QED OBVIOUS}\)

(2.15. CASE \textit{NextSealTs})

\(\text{(3) USE } \textit{NextSealTs}\)

\(\text{(3) USE DEF } \textit{NextSealTs}\)

\(\text{(3) USE DEF } \textit{Pcrx}\)

\(\text{(3) } \textit{sealPcr} \in \textit{Pcr} \text{ BY DEF } \textit{InvType}\)

\(\text{(3) } \textit{SealReboot} \in \textit{Pcr} \text{ BY ThmSealRebootIsPcr}\)

\(\text{(3) QED BY ThmPcrExtendFromEqOrNotleq}\)

(2.16. CASE \textit{NextEnterSemChkpt})

\(\text{(3) USE } \textit{NextEnterSemChkpt}\)

\(\text{(3) USE DEF } \textit{NextEnterSemChkpt}\)

\(\text{(3) QED OBVIOUS}\)

(2.17. CASE \textit{NextSemChkpt1 WhenCorrect})

\(\text{(3) USE } \textit{NextSemChkpt1 WhenCorrect}\)

\(\text{(3) USE DEF } \textit{NextSemChkpt1 WhenCorrect}\)

\(\text{(3) QED OBVIOUS}\)

(2.18. CASE \textit{NextSemChkpt1 WhenIncorrect})

\(\text{(3) USE } \textit{NextSemChkpt1 WhenIncorrect}\)

\(\text{(3) USE DEF } \textit{NextSemChkpt1 WhenIncorrect}\)

\(\text{(3) QED OBVIOUS}\)

(2.19. CASE \textit{NextSemChkpt2})

\(\text{(3) USE } \textit{NextSemChkpt2}\)

\(\text{(3) USE DEF } \textit{NextSemChkpt2}\)

\(\text{(3) QED OBVIOUS}\)

(2.20. CASE \textit{NextSemChkpt3})

\(\text{(3) USE } \textit{NextSemChkpt3}\)

\(\text{(3) USE DEF } \textit{NextSemChkpt3}\)

\(\text{(3) QED OBVIOUS}\)

(2.21. CASE \textit{NextSemChkpt4})

\(\text{(3) USE } \textit{NextSemChkpt4}\)

\(\text{(3) USE DEF } \textit{NextSemChkpt4}\)

\(\text{(3) QED OBVIOUS}\)

(2.22. CASE \textit{NextSemChkpt5})

\(\text{(3) USE } \textit{NextSemChkpt5}\)

\(\text{(3) USE DEF } \textit{NextSemChkpt5}\)
3. QED OBVIOUS

2. QED
   BY (2) 1,
   (2) 2, (2) 3, (2) 4, (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, (2) 21, (2) 22
   DEF Next
   (1) QED OBVIOUS

It is an invariant of the specification.

THEOREM ThmInvUnforgeableSealReboot \[ \triangleq \]
   Spec \( \Rightarrow \) \( \Box \) InvUnforgeableSealReboot

PROOF
   (1) Init \( \Rightarrow \) InvUnforgeableSealReboot
      BY ThmInitInvUnforgeableSealReboot
   (1) InvUnforgeableSealReboot \( \land \) [Next]\( vars \) \( \Rightarrow \) InvUnforgeableSealReboot'
      BY ThmNextInvUnforgeableSealReboot
   (1) USE DEF Spec
   (1) QED

PROOF OF INVARIANT InvProperLastExtension

It holds in the initial state.

THEOREM ThmInitInvProperLastExtension \[ \triangleq \]
   Init \( \Rightarrow \) InvProperLastExtension

PROOF
   (1) HAVE Init
   (1) InvType
      BY ThmInitInvType
   (1) QED
      BY DEF InvProperLastExtension, Init

If it holds in the current state, and we perform a Next action, then it will hold in the next state.

THEOREM ThmNextInvProperLastExtension \[ \triangleq \]
   InvProperLastExtension \( \land \) [Next]\( vars \) \( \Rightarrow \) InvProperLastExtension'

PROOF

(1) HAVE InvProperLastExtension \land [Next]_{\text{vars}}
(1) USE DEF InvProperLastExtension

(1) InvType' BY ThmNextInvType
(1) InvProperLastExtension' goal'

Wow, this is an easy one.

(2) USE DEF vars
(2) USE DEF NextObtainAccess
(2) USE DEF NextProveRevoke
(2) USE DEF NextReboot
(2) USE DEF NextForgetSealTs
(2) USE DEF NextExtendAppPcr
(2) USE DEF NextExtendSemPcr
(2) USE DEF NextExtendSealPcr
(2) USE DEF NextIncBootCtr
(2) USE DEF NextEnterSemRecover
(2) USE DEF NextSemRecover1 WhenCorrect
(2) USE DEF NextSemRecover1 WhenIncorrect
(2) USE DEF NextSemRecover2
(2) USE DEF NextSemRecover3
(2) USE DEF NextSealTs
(2) USE DEF NextEnterSemRecover
(2) USE DEF NextSemChkpt1 WhenCorrect
(2) USE DEF NextSemChkpt1 WhenIncorrect
(2) USE DEF NextSemChkpt2
(2) USE DEF NextSemChkpt3
(2) USE DEF NextSemChkpt4
(2) USE DEF NextSemChkpt5

(2) QED BY DEF Next
(1) QED OBVIOUS

It is an invariant of the specification.

THEOREM ThmInvProperLastExtension \triangleq Spec \Rightarrow \Box \text{InvProperLastExtension}
PROOF

(1) Init \Rightarrow InvProperLastExtension BY ThmInitInvProperLastExtension
(1) InvProperLastExtension \land [Next]_{\text{vars}} \Rightarrow InvProperLastExtension'
BY ThmNextInvProperLastExtension
(1) USE DEF Spec
(1) QED
PROOF OF INVARIANT $InvOneLog$

It holds in the initial state.

THEOREM $ThmInitInvOneLog \iff$

\[ Init \Rightarrow InvOneLog \]

PROOF

(1) HAVE $Init$

(1) $InvType$ BY $ThmInitInvType$

(1) $InvSignedTsLeqBoot$ BY $ThmInitInvSignedTsLeqBoot$

(1) $InvInSemProtect$ BY $ThmInitInvInSemProtect$

(1) $InvUnforgeableSemHappy$ BY $ThmInitInvUnforgeableSemHappy$

(1) $InvProperLastExtension$ BY $ThmInitInvProperLastExtension$

(1) USE DEF $Init$

(1) USE DEF $InitNv$

(1) USE DEF $PcIDLE$

(1) USE DEF $PcRECOV1$, $PcRECOV2$, $PcRECOV3$

(1) USE DEF $PcCHKPT1$, $PcCHKPT2$, $PcCHKPT3$, $PcCHKPT4$, $PcCHKPT5$

(1) $LogInNv$ BY DEF $LogInNv$

(1) $\neg LogInApp$

(2) $semPcr \neq SemHappy$

(3) $semPcr.init \neq SemHappy.init$

(4) $semPcr.init \neq SemProtect.init$

(5) USE DEF $SemReboot$

(5) USE DEF $SemProtect$

(5) QED BY $AssSemProtect$

(4) USE DEF $SemHappy$

(4) QED BY DEF $PcrExtend$

(3) QED BY DEF $Pcr$

(2) QED BY DEF $LogInApp$

(1) $\neg LogInTs \land AllCurrentTs = \{\}$

(2) $\neg CheckTsIsCurrent(chkptts)$

(3) QED BY $ThmNullTsIsntSignedTs$

(2) USE DEF $AllCurrentTs$

(2) QED BY DEF $LogInTs$

(1) $InvOneLog!goal!obtains$ BY DEF $IsOnLog$

(1) $InvOneLog!goal!revokes$ BY DEF $IsOnLog$

(1) $InvVerifiableRevocation$ BY DEF $InvVerifiableRevocation$

(1) QED BY DEF $InvOneLog$
If it holds in the current state, and we perform a Next action, then it will hold in the next state.

**THEOREM**  \( \text{ThmNextInvOneLog} \)  
\[ \text{InvOneLog} \land [\text{Next}]_{\text{vars}} \Rightarrow \text{InvOneLog}' \]

**PROOF**

(1) **HAVE** \( \text{InvOneLog} \land [\text{Next}]_{\text{vars}} \)  

(1) \( \text{InvType} \)  **BY** DEF \( \text{InvOneLog} \)  
(1) \( \text{InvSignedTsLeqBoot} \)  **BY** DEF \( \text{InvOneLog} \)  
(1) \( \text{InvInSemProtect} \)  **BY** DEF \( \text{InvOneLog} \)  
(1) \( \text{InvUnforgeableSemHappy} \)  **BY** DEF \( \text{InvOneLog} \)  
(1) \( \text{InvUnforgeableSealReboot} \)  **BY** DEF \( \text{InvOneLog} \)  
(1) \( \text{InvProperLastExtension} \)  **BY** DEF \( \text{InvOneLog} \)  

(1) \( \text{InvType}' \)  **BY** ThmNextInvType  
(1) \( \text{InvSignedTsLeqBoot}' \)  **BY** ThmNextInvSignedTsLeqBoot  
(1) \( \text{InvInSemProtect}' \)  **BY** ThmNextInvInSemProtect  
(1) \( \text{InvUnforgeableSemHappy}' \)  **BY** ThmNextInvUnforgeableSemHappy  
(1) \( \text{InvUnforgeableSealReboot}' \)  **BY** ThmNextInvUnforgeableSealReboot  
(1) \( \text{InvProperLastExtension}' \)  **BY** ThmNextInvProperLastExtension  

(1) \( \text{InvOneLog}' \)  **goal'**  

(2) USE DEF \( \text{PcIDLE} \)  
(2) USE DEF \( \text{PcRECOV1}, \text{PcRECOV2}, \text{PcRECOV3} \)  
(2) USE DEF \( \text{PcCHKPT1}, \text{PcCHKPT2}, \text{PcCHKPT3}, \text{PcCHKPT4}, \text{PcCHKPT5} \)  

**Stutter step.**

(2) 1. **CASE** \( \text{vars}' = \text{vars} \)  
(3) USE (2) 1  
(3) USE DEF \( \text{vars} \)  
(3) UNCHANGED \( \text{CheckTsIsCurrent}(\text{chkptts}) \)  **BY** DEF \( \text{CheckTsIsCurrent} \)  
(3) UNCHANGED \( \text{AllCurrentTs} \)  
(4) USE DEF \( \text{AllCurrentTs} \)  
(4) USE DEF \( \text{CheckTsIsCurrent} \)  
(4) QED **BY** DEF \( \text{LogInTs} \)  

(3) UNCHANGED \( \text{CurrentTsLog} \)  **BY** DEF \( \text{CurrentTsLog} \)  
(3) UNCHANGED \( \text{LogInNu} \)  **BY** DEF \( \text{LogInNu} \)  
(3) UNCHANGED \( \text{LogInApp} \)  **BY** DEF \( \text{LogInApp} \)  
(3) UNCHANGED \( \text{LogInTs} \)  **BY** DEF \( \text{LogInTs} \)  
(3) \( \text{InvOneLog'} \)  **goal' obtains'**  **BY** DEF \( \text{IsOnLog}, \text{InvOneLog} \)  
(3) \( \text{InvOneLog'} \)  **goal' revokes'**  **BY** DEF \( \text{IsOnLog}, \text{InvOneLog} \)  
(3) \( \text{InvVerifiableRevocation}' \)  **BY** DEF \( \text{InvVerifiableRevocation}, \text{InvOneLog} \)  
(3) QED **BY** DEF \( \text{InvOneLog} \)
NextObtainAccess or NextProveRevoke

(2). CASE NextObtainAccess \lor NextProveRevoke
   (3) USE (2).2
   (3) USE DEF NextObtainAccess
   (3) USE DEF NextProveRevoke
   (3) UNCHANGED CheckTsIsCurrent(chkpts) BY DEF CheckTsIsCurrent
   (3) UNCHANGED AllCurrentTs
       (4) USE DEF AllCurrentTs
       (4) USE DEF CheckTsIsCurrent
       (4) QED BY DEF LogInTs
   (3) UNCHANGED CurrentTsLog BY DEF CurrentTsLog
   (3) UNCHANGED LogInNu BY DEF LogInNu
   (3) UNCHANGED LogInApp BY DEF LogInApp
   (3) UNCHANGED LogInTs BY DEF LogInTs
   (3) LogInApp
       (4) CASE NextObtainAccess
           NextObtainAccess is predicated on the fact that the log is in the application pcr.
           This depends on
           \( \land \text{BugObtainAccessNoCheckHappy} = \text{FALSE} \)
           \( \land \text{BugObtainAccessNoCheckSeal} = \text{FALSE} \)

       (5) BugObtainAccessNoCheckHappy = FALSE BY DEF BugObtainAccessNoCheckHappy
       (5) BugObtainAccessNoCheckSeal = FALSE BY DEF BugObtainAccessNoCheckSeal
       (5) QED BY DEF LogInApp

   (4) CASE NextProveRevoke
       NextProveRevoke is predicated on the fact that the log is in the application pcr.
       This depends on
       \( \land \text{BugProveRevokeNoCheckHappy} = \text{FALSE} \)
       \( \land \text{BugProveRevokeNoCheckSeal} = \text{FALSE} \)

       (5) BugProveRevokeNoCheckHappy = FALSE BY DEF BugProveRevokeNoCheckHappy
       (5) BugProveRevokeNoCheckSeal = FALSE BY DEF BugProveRevokeNoCheckSeal
       (5) QED BY DEF LogInApp
       (4) QED OBVIOUS
   (3) InvOneLog' goal! obtains' \land InvOneLog' goal! revokes'

   Since the log is in the application pcr, putting a copy of the application pcr into obtains or into revokes preserves the invariant that everything in obtains \cup revokes can reach the log.
   (4) PcrLeq(appPcr, appPcr) BY ThmPcrLeqIsReflexive DEF InvType
   (4) QED BY DEF IsOnLog, InvOneLog
   (3) InvVerifiableRevocation'

   Since we only add an element to obtains \cup revokes when the log is in the application pcr, we know that all elements in obtains \cup revokes in the new state must be on the log, which we can check as \( \leq \text{app pcr} \).

   So we proceed with proof by contradiction. Assuming that verifiable deletion will be violated in the new state, we pick the \( o \in obtains \) and \( r \in revokes \) whose PcrPrior's are the same. But since both \( o \) and \( r \) must be \( \leq \text{app pcr} \), this means that their last extension must be the same. This contradicts the assumption that OBTAIN is different from REVOKE.

   (4) CASE InvVerifiableRevocation' OBVIOUS
   (4) CASE \neg InvVerifiableRevocation'
\(\langle 5 \rangle\) PICK \(o \in \text{obtains}'\), \(r \in \text{revoke}'\) : \(\text{PcrPrior}(o) = \text{PcrPrior}(r)\)

BY DEF \text{InvVerifiableRevocation}

\(\langle 5 \rangle\) DEFINE \(p \triangleq \text{PcrPrior}(o)\)

\(\langle 5 \rangle\) DEFINE \(xo \triangleq \text{PcrLastExtension}(o)\)

\(\langle 5 \rangle\) DEFINE \(xr \triangleq \text{PcrLastExtension}(r)\)

\(\langle 5 \rangle\) \(p \in \text{Pcr}\) BY \text{ThmPcrPriorIsPcr}

\(\langle 5 \rangle\) \(xo \in \text{Pcrx}\) BY DEF \text{InvType}, \text{InvProperLastExtension}

\(\langle 5 \rangle\) \(xr \in \text{Pcrx}\) BY DEF \text{Pcrx}

\(\langle 5 \rangle\) \(xo = xr\)

The prover needs a lot of help to focus its attention.

\(\langle 6 \rangle\) HIDE DEF \(p\)

\(\langle 6 \rangle\) HIDE DEF \(xo\)

\(\langle 6 \rangle\) HIDE DEF \(xr\)

\(\langle 6 \rangle\) \(\text{appPcr}' \in \text{Pcr}\) BY DEF \text{InvType}

\(\langle 6 \rangle\) QED BY \text{ThmPcrExtendLegAnticollision}

\(\langle 5 \rangle\) \(\text{PcrxOBTAIN} \neq \text{PcrxREVOKE}'\) BY DEF \text{AssObtainNeqRevoke}

\(\langle 5 \rangle\) QED OBVIOUS

\(\langle 4 \rangle\) QED OBVIOUS

\(\langle 3 \rangle\) QED BY DEF \text{InvOneLog}

\hline*

NextReboot

\(\langle 2 \rangle\) CASE NextReboot

\(\langle 3 \rangle\) USE NextReboot

\(\langle 3 \rangle\) USE DEF NextReboot

\(\langle 3 \rangle\) UNCHANGED \text{LogInNv} \quad \text{BY DEF \text{LogInNv}}

Cancels \text{SemHappy} if we had it, which erases any log that had been in the application pcr.

\(\langle 3 \rangle\) \(\neg \text{LogInApp}'\)

\(\langle 4 \rangle\) \(\text{semPcr}' \neq \text{SemHappy}\)

\(\langle 5 \rangle\) \(\text{semPcr}'.init \neq \text{SemHappy}.init\)

\(\langle 6 \rangle\) \(\text{semPcr}'.init \neq \text{SemProtect}.init\)

\(\langle 7 \rangle\) USE DEF \text{SemReboot}

\(\langle 7 \rangle\) USE DEF \text{SemProtect}

\(\langle 7 \rangle\) USE DEF \text{PcrInit}

\(\langle 7 \rangle\) QED BY \text{AssSemProtect}

\(\langle 6 \rangle\) USE DEF \text{SemHappy}

\(\langle 6 \rangle\) QED BY DEF \text{PcrExtend}

\(\langle 5 \rangle\) QED BY DEF \text{Pcr}

\(\langle 4 \rangle\) QED BY DEF \text{LogInApp}
Overwrites `chkptts` with an unsigned `ts`, which might erase a log that had been in the seal attestations.

\[\langle 3 \rangle \text{LogInTs}' \Rightarrow \text{LogInTs}\]

\[\langle 4 \rangle \text{HAVE LogInTs}' \]

\[\langle 4 \rangle \text{AllCurrentTs}' \neq \{\} \text{BY DEF LogInTs}\]

\[\langle 4 \rangle \text{USE DEF AllCurrentTs}\]

\[\langle 4 \rangle \text{USE DEF CheckTsIsCurrent}\]

\[\langle 4 \rangle \neg \text{CheckTsIsCurrent(chkptts)}' \text{BY ThmNullTsIsntSignedTs}\]

\[\langle 4 \rangle \exists ts \in tsvalues': \text{CheckTsIsCurrent(ts)} \text{OBVIOUS}\]

\[\langle 4 \rangle \text{QED BY DEF LogInTs}\]

Any remaining current seal attestations existed previously, so they must contain the same log.

\[\langle 3 \rangle \forall ts_1, ts_2 \in \text{AllCurrentTs'}: ts_1.\text{appPcr} = ts_2.\text{appPcr}\]

\[\langle 4 \rangle \text{TAKIE ts}_1, ts_2 \in \text{AllCurrentTs'}\]

\[\langle 4 \rangle \text{USE DEF AllCurrentTs}\]

\[\langle 4 \rangle \text{USE DEF CheckTsIsCurrent}\]

\[\langle 4 \rangle \neg \text{CheckTsIsCurrent(chkptts)}' \text{BY ThmNullTsIsntSignedTs}\]

\[\langle 4 \rangle ts_1 \in \text{AllCurrentTs} \text{OBVIOUS}\]

\[\langle 4 \rangle ts_2 \in \text{AllCurrentTs} \text{OBVIOUS}\]

\[\langle 4 \rangle \text{QED BY DEF InvOneLog}\]

If there are any remaining current seal attestations, the log in them has to be the same as before.

\[\langle 3 \rangle \text{LogInTs}' \Rightarrow \text{UNCHANGED CurrentTsLog}\]

\[\langle 4 \rangle \text{HAVE LogInTs}'\]

\[\langle 4 \rangle \exists ts \in \text{AllCurrentTs'}: ts \in \text{AllCurrentTs}\]

\[\langle 5 \rangle \text{USE DEF LogInTs}\]

\[\langle 5 \rangle \text{USE DEF AllCurrentTs}\]

\[\langle 5 \rangle \text{USE DEF CheckTsIsCurrent}\]

\[\langle 5 \rangle \neg \text{CheckTsIsCurrent(chkptts)}' \text{BY ThmNullTsIsntSignedTs}\]

\[\langle 5 \rangle \forall ts \in \text{AllCurrentTs'}: ts \in \text{AllCurrentTs} \text{OBVIOUS}\]

\[\langle 5 \rangle \text{QED OBVIOUS}\]

\[\langle 4 \rangle \forall ts_1 \in \text{AllCurrentTs}':\]

\[\forall ts_0 \in \text{AllCurrentTs}: ts_0.\text{appPcr} = ts_1.\text{appPcr} \]

\[\text{BY DEF InvOneLog}\]

\[\langle 4 \rangle \text{QED BY DEF CurrentTsLog}\]

\[\langle 3 \rangle \text{InvOneLog}' \text{goal} \text{ obtains}' \text{BY DEF IsOnLog, InvOneLog}\]

\[\langle 3 \rangle \text{InvOneLog}' \text{goal} \text{ revokes}' \text{BY DEF IsOnLog, InvOneLog}\]

\[\langle 3 \rangle \text{InvVerifiableRevocation}' \text{BY DEF InvVerifiableRevocation, InvOneLog}\]

\[\langle 3 \rangle \text{QED BY DEF InvOneLog}\]

\[\text{NextForgetSealTs}\]

\[\langle 2 \rangle \text{4. CASE NextForgetSealTs}\]

\[\text{USE NextForgetSealTs}\]

\[\text{USE DEF NextForgetSealTs}\]

\[\text{UNCHANGED LogInNv} \text{BY DEF LogInNv}\]

\[\text{UNCHANGED LogInApp} \text{BY DEF LogInApp}\]
UNCHANGED \( \text{CheckTsIsCurrent}(\text{chkptts}) \) \( \) BY DEF \( \text{CheckTsIsCurrent} \)

Forgets a seal attestation, which might erase a log that had been in the seal attestations.

\( \langle 3 \rangle \) \( \text{LogInTs}' \Rightarrow \text{LogInTs} \)

(4) HAVE \( \text{LogInTs}' \)

(4) \( \text{AllCurrentTs}' \neq \{\} \) \( \) BY DEF \( \text{LogInTs} \)

(4) USE DEF \( \text{AllCurrentTs} \)

(4) USE DEF \( \text{CheckTsIsCurrent} \)

(4) QED BY DEF \( \text{LogInTs} \)

Any remaining current seal attestations existed previously, so they must contain the same log.

\( \langle 3 \rangle \forall ts_1, ts_2 \in \text{AllCurrentTs}' : ts_1.\text{appPcr} = ts_2.\text{appPcr} \)

(4) TAKE \( ts_1, ts_2 \in \text{AllCurrentTs}' \)

(4) USE DEF \( \text{AllCurrentTs} \)

(4) USE DEF \( \text{CheckTsIsCurrent} \)

(4) \( ts_1 \in \text{AllCurrentTsORBVIOUS} \)

(4) \( ts_2 \in \text{AllCurrentTsORBVIOUS} \)

(4) QED BY DEF \( \text{InvOneLog} \)

If \( \text{chkptts} \) contains a current seal attestation, then the log is in the seal attestations.

\( \langle 3 \rangle \text{CheckTsIsCurrent}(\text{chkptts})' \Rightarrow \text{LogInTs}' \)

(4) USE DEF \( \text{LogInTs} \)

(4) USE DEF \( \text{AllCurrentTs} \)

(4) USE DEF \( \text{CheckTsIsCurrent} \)

(4) QED OBVIOUS

If there are any remaining current seal attestations, the log in them has to be the same as before.

\( \langle 3 \rangle \text{LogInTs}' \Rightarrow \text{UNCHANGED \text{CurrentTsLog}} \)

(4) HAVE \( \text{LogInTs}' \)

(4) \( \exists ts \in \text{AllCurrentTs}' : ts \in \text{AllCurrentTs} \)

\( \langle 5 \rangle \) USE DEF \( \text{LogInTs} \)

\( \langle 5 \rangle \) USE DEF \( \text{AllCurrentTs} \)

\( \langle 5 \rangle \) USE DEF \( \text{CheckTsIsCurrent} \)

\( \langle 5 \rangle \forall ts \in \text{AllCurrentTs}' : ts \in \text{AllCurrentTsORBVIOUS} \)

\( \langle 5 \rangle \) QED OBVIOUS

(4) \( \forall ts_1 \in \text{AllCurrentTs}' : \)

\( \forall ts_0 \in \text{AllCurrentTs} : \)

\( ts_1.\text{appPcr} = ts_0.\text{appPcr} \)

BY DEF \( \text{InvOneLog} \)

(4) QED BY DEF \( \text{CurrentTsLog} \)

\( \langle 3 \rangle \text{InvOneLog}' \) \( \) goal \( \) obtains \( \) \( \) BY DEF \( \text{IsOnLog}, \text{InvOneLog} \)

\( \langle 3 \rangle \text{InvOneLog}' \) \( \) goal \( \) revokes \( \) \( \) BY DEF \( \text{IsOnLog}, \text{InvOneLog} \)

\( \langle 3 \rangle \text{InvVerifiableRevocation}' \) \( \) \( \) BY DEF \( \text{InvVerifiableRevocation}, \text{InvOneLog} \)

\( \langle 3 \rangle \) QED BY DEF \( \text{InvOneLog} \)

\( \) NextExtendAppPcr
5. CASE NextExtendAppPcr
   (3) use NextExtendAppPcr
   (3) USE DEF NextExtendAppPcr
   (3) UNCHANGED CheckTsIsCurrent(chkptts) BY DEF CheckTsIsCurrent
   (3) UNCHANGED AllCurrentTs
   (4) USE DEF AllCurrentTs
   (4) USE DEF CheckTsIsCurrent
   (4) QED BY DEF LogInTs
   (3) UNCHANGED CurrentTsLog BY DEF CurrentTsLog
   (3) UNCHANGED LogInNv BY DEF LogInNv
   (3) UNCHANGED LogInApp BY DEF LogInApp
   (3) UNCHANGED LogInTs
   (4) USE DEF AllCurrentTs
   (4) USE DEF CheckTsIsCurrent
   (4) QED BY DEF LogInTs
   (3) InvVerifiableRevocation' BY DEF InvVerifiableRevocation, InvOneLog
   (3) InvOneLog!goal!obtains' ∧ InvOneLog!goal!revokes'
   (4) CASE ¬LogInApp
      (5) USE DEF LogInApp
      (5) USE DEF IsOnLog, InvOneLog
      NextExtendAppPcr is predicated on not being in sem, so none of the sem clauses apply.
   (5) USE DEF InSem
   (5) QED OBVIOUS
   (4) CASE LogInApp
      (5) USE DEF LogInApp
   (4) QED BY DEF LogInApp
If the log is in the application pcr, extending the application pcr preserves the fact that all entries in obtains ∪ revokes can reach it.
   (5) ∀ p ∈ obtains ∪ revokes : LogInApp ⇒ PcrLeq(p, appPcr')
      (6) UNCHANGED (obtains ∪ revokes) OBVIOUS
      (6) TAKE p ∈ obtains ∪ revokes
      (6) HAVE LogInApp
      (6) PcrLeq(p, appPcr') BY DEF IsOnLog, InvOneLog
      (6) PcrLeq(appPcr, appPcr') BY ThmPcrExtendLeq DEF InvType
      (6) QED BY ThmPcrLeqIsTransitive DEF InvType
      (5) QED BY DEF IsOnLog, InvOneLog
      (4) QED OBVIOUS
   (3) QED BY DEF InvOneLog

6. CASE NextExtendSemPcr
   (3) use NextExtendSemPcr
   (3) USE DEF NextExtendSemPcr
   (3) UNCHANGED CheckTsIsCurrent(chkptts) BY DEF CheckTsIsCurrent
   (3) UNCHANGED AllCurrentTs
   (4) USE DEF AllCurrentTs
\( \langle 4 \rangle \) USE DEF CheckTsIsCurrent
\( \langle 4 \rangle \) QED BY DEF LogInTs
\( \langle 3 \rangle \) UNCHANGED CurrentTsLog BY DEF CurrentTsLog
\( \langle 3 \rangle \) UNCHANGED LogInNu BY DEF LogInNu

Cancels \textit{SemHappy} if we had it, which erases any log that had been in the application pcr.

\( \langle 3 \rangle \) \textbf{¬LogInApp'}
\( \langle 4 \rangle \) \textit{semPcr'} \neq \textit{SemHappy}
\textit{semPcr} = \textit{SemHappy} \lor \neg \textit{PcrLeq}(\textit{semPcr}, \textit{SemHappy})
\textit{by DEF InvUnforgeableSemHappy}
\textit{semPcr} \in \textit{Pcr} \textit{by DEF InvType}
\textit{SemHappy} \in \textit{Pcr} \textit{by ThmSemHappyIsPcr}
\textit{¬PcrLeq}(\textit{semPcr'}, \textit{SemHappy}) \textit{by ThmPcrExtendFromEqOrNotleq}
\( \langle 5 \rangle \) QED BY ThmPcrLeqIsReflexive
\( \langle 4 \rangle \) QED BY DEF LogInApp
\( \langle 3 \rangle \) UNCHANGED LogInTs
\( \langle 3 \rangle \) USE DEF AllCurrentTs
\( \langle 4 \rangle \) USE DEF CheckTsIsCurrent
\( \langle 4 \rangle \) QED BY DEF LogInTs
\( \langle 3 \rangle \) InvOneLog !goal! obtains' \textit{by DEF IsOnLog, InvOneLog}
\( \langle 3 \rangle \) InvOneLog !goal! revokes' \textit{by DEF IsOnLog, InvOneLog}
\( \langle 3 \rangle \) InvVerifiableRevocation' \textit{by DEF InvVerifiableRevocation, InvOneLog}
\( \langle 3 \rangle \) QED BY DEF InvOneLog

NextExtendSealPcr
\( \langle 2 \rangle \) 7. CASE NextExtendSealPcr
\( \langle 3 \rangle \) USE NextExtendSealPcr
\( \langle 3 \rangle \) USE DEF NextExtendSealPcr
\( \langle 3 \rangle \) UNCHANGED CheckTsIsCurrent(chkptts) BY DEF CheckTsIsCurrent
\( \langle 3 \rangle \) UNCHANGED AllCurrentTs
\( \langle 4 \rangle \) USE DEF AllCurrentTs
\( \langle 4 \rangle \) USE DEF CheckTsIsCurrent
\( \langle 4 \rangle \) QED BY DEF LogInTs
\( \langle 3 \rangle \) UNCHANGED CurrentTsLog BY DEF CurrentTsLog
\( \langle 3 \rangle \) UNCHANGED LogInNu BY DEF LogInNu

Cancels \textit{SealReboot} if we had it, which erases any log that had been in the application pcr.

\( \langle 3 \rangle \) \textbf{¬LogInApp'}
\( \langle 4 \rangle \) \textit{sealPcr'} \neq \textit{SealReboot}
\textit{sealPcr} = \textit{SealReboot} \lor \neg \textit{PcrLeq}(\textit{sealPcr}, \textit{SealReboot})
\textit{by DEF InvUnforgeableSealReboot}
\textit{sealPcr} \in \textit{Pcr} \textit{by DEF InvType}
\textit{SealReboot} \in \textit{Pcr} \textit{by ThmSealRebootIsPcr}
\textit{¬PcrLeq}(\textit{sealPcr'}, \textit{SealReboot}) \textit{by ThmPcrExtendFromEqOrNotleq}
\( \langle 5 \rangle \) QED BY ThmPcrLeqIsReflexive
\( \langle 4 \rangle \) QED BY DEF LogInApp
⟨3⟩ UNCHANGED LogInTs
(4) USE DEF AllCurrentTs
(4) USE DEF CheckTsIsCurrent
(4) QED BY DEF LogInTs

⟨3⟩ InvOneLog \text{!'goal!' obtains} \text{'} BY DEF IsOnLog, InvOneLog
⟨3⟩ InvOneLog \text{!'goal!' revokes} \text{'} BY DEF IsOnLog, InvOneLog
⟨3⟩ InvVerifiableRevocation \text{'} BY DEF InvVerifiableRevocation, InvOneLog
⟨3⟩ QED BY DEF InvOneLog

NextIncBootCtr

(2) 8. CASE NextIncBootCtr
(3) USE NextIncBootCtr
(3) USE DEF NextIncBootCtr
(3) UNCHANGED LogInNu BY DEF LogInNu
(3) UNCHANGED LogInApp BY DEF LogInApp

Since no signed ts seal can have a bootCtr greater than the current bootCtr, incrementing bootCtr erases any log that had been in a seal attestation.

⟨3⟩ ¬LogInTs'
(4) USE DEF AllCurrentTs
(4) USE DEF CheckTsIsCurrent
(4) ∀ ts ∈ tsvalues' ∪ {chkptts'} : ts ∈ SignedTs ⇒ ts.bootCtr ≠ bootCtr'
(5) TAKE ts ∈ tsvalues' ∪ {chkptts'}
(5) ts ∈ tsvalues ∪ {chkptts} OBVIOUS
(5) HAVE ts ∈ SignedTs
(5) ts.bootCtr ≤ bootCtr' BY DEF InvSignedTsLeqBoot
(5) ts.bootCtr ∈ Nat BY DEF SignedTs
(5) bootCtr ∈ Nat BY DEF InvType
(5) bootCtr' ∈ Nat BY DEF InvType
(5) bootCtr < bootCtr' BY ThmNatInc
(5) ts.bootCtr < bootCtr' BY ThmNatLeqLt
(5) QED BY ThmNatLeqXorGt, ThmNatLeqIsReflexive
(4) QED BY DEF LogInTs

Erasers all current ts seal attestations.

⟨3⟩ AllCurrentTs' = {} ∧ ¬CheckTsIsCurrent(chkptts)'
(4) USE DEF AllCurrentTs
(4) USE DEF CheckTsIsCurrent
(4) QED BY DEF LogInTs

⟨3⟩ USE Def InSem
⟨3⟩ InvOneLog \text{!'goal!' obtains} \text{'} BY DEF IsOnLog, InvOneLog
⟨3⟩ InvOneLog \text{!'goal!' revokes} \text{'} BY DEF IsOnLog, InvOneLog
⟨3⟩ InvVerifiableRevocation \text{'} BY DEF InvVerifiableRevocation, InvOneLog
⟨3⟩ QED BY DEF InvOneLog
NextEnterSemRecov

(2) 9. CASE NextEnterSemRecov
    (3) USE NextEnterSemRecov
    (3) USE DEF NextEnterSemRecov
    (3) USE InSem
    (3) UNCHANGED CheckTsIsCurrent(chkptts) BY DEF CheckTsIsCurrent
    (3) UNCHANGED AllCurrentTs
      (4) USE DEF AllCurrentTs
      (4) USE DEF CheckTsIsCurrent
      (4) QED BY DEF LogInTs
    (3) UNCHANGED CurrentTsLog BY DEF CurrentTsLog
    (3) UNCHANGED LogInNu BY DEF LogInNu

Cancels SemHappy if we had it, which erases any log that had been in the application pcr.

(3) ¬LogInApp'
      (3) SemPcr' ≠ SemHappy
        (5) USE DEF SemHappy
        (5) USE DEF SemProtect
        (5) USE DEF Pcri
        (5) USE DEF Pcrx
        (5) USE ThmPcrInitIsPcr
        (5) USE ThmPcrExtendIsPcr
        (5) PcrLeq(SemProtect, SemHappy) BY ThmPcrExtendLeq
        (5) ¬PcrLeq(SemHappy, SemProtect) BY ThmPcrExtendSelfUnreachable
        (5) QED BY ThmPcrLeqIsAntisymmetric
    (4) QED BY DEF LogInApp
    (3) UNCHANGED LogInTs
      (4) USE DEF AllCurrentTs
      (4) USE DEF CheckTsIsCurrent
      (4) QED BY DEF LogInTs
    (3) InvOneLog (goal obtains)' BY DEF IsOnLog, InvOneLog
    (3) InvOneLog (goal revokes)' BY DEF IsOnLog, InvOneLog
    (3) InvVerifiableRevocation' BY DEF InvVerifiableRevocation, InvOneLog
    (3) QED BY DEF InvOneLog

NextSemRecov1WhenCorrect

(2) 10. CASE NextSemRecov1WhenCorrect
    (3) USE NextSemRecov1WhenCorrect
    (3) USE DEF NextSemRecov1WhenCorrect
    (3) UNCHANGED CheckTsIsCurrent(chkptts) BY DEF CheckTsIsCurrent
    (3) UNCHANGED AllCurrentTs
      (4) USE DEF AllCurrentTs
      (4) USE DEF CheckTsIsCurrent
      (4) QED BY DEF LogInTs
    (3) UNCHANGED CurrentTsLog BY DEF CurrentTsLog
EnterSemRecovPredicate guarantees that the log is in the \( \text{nv} \) ram.

This depends on \( \text{BugRecovNoCheckCur} \triangleq \text{FALSE} \)

(3) \( \text{LogInNv} \)
(4) USE DEF \( \text{EnterSemRecovPredicate} \)
(4) \( \text{BugRecovNoCheckCur} = \text{FALSE} \) BY DEF \( \text{BugRecovNoCheckCur} \)
(4) QED BY DEF \( \text{LogInNv} \)

EnterSemRecovPredicate guarantees that the application pcr equals the log saved in the \( \text{nv} \) ram.

This depends on \( \text{BugRecovNoCheckApp} \triangleq \text{FALSE} \)

(3) \( \text{appPcr} = \text{nv} \cdot \text{appPcr} \)
(4) USE DEF \( \text{EnterSemRecovPredicate} \)
(4) \( \text{BugRecovNoCheckApp} = \text{FALSE} \) BY DEF \( \text{BugRecovNoCheckApp} \)
(4) QED OBVIOUS

(3) \( \text{InvOneLog} \cdot \text{goal} \cdot \text{obtains}' \) BY DEF \( \text{IsOnLog}, \text{InvOneLog} \)
(3) \( \text{InvOneLog} \cdot \text{goal} \cdot \text{revokes}' \) BY DEF \( \text{IsOnLog}, \text{InvOneLog} \)
(3) \( \text{InvVerifiableRevocation}' \) BY DEF \( \text{InvVerifiableRevocation}, \text{InvOneLog} \)
(3) QED BY DEF \( \text{InvOneLog} \)

NextSemRecov1WhenIncorrect

(2) CASE \( \text{NextSemRecov1WhenIncorrect} \)
(3) USE \( \text{NextSemRecov1WhenIncorrect} \)
(3) USE DEF \( \text{NextSemRecov1WhenIncorrect} \)
(3) UNCHANGED \( \text{CheckTsIsCurrent(chkptts)} \) BY DEF \( \text{CheckTsIsCurrent} \)
(3) UNCHANGED \( \text{AllCurrentTs} \)
(4) USE DEF \( \text{AllCurrentTs} \)
(4) USE DEF \( \text{CheckTsIsCurrent} \)
(4) QED BY DEF \( \text{LogInTs} \)
(3) UNCHANGED \( \text{CurrentTsLog} \) BY DEF \( \text{CurrentTsLog} \)
(3) UNCHANGED \( \text{LogInNv} \) BY DEF \( \text{LogInNv} \)

Extending sem pcr with Unhappy results in something other than \( \text{SemHappy} \), which indicates that the log is not in the application pcr.

(3) \( \neg \text{LogInApp}' \)
(4) \( \text{semPcr}' \neq \text{SemHappy} \)
(5) USE DEF \( \text{SemHappy} \)
(5) USE DEF \( \text{SemProtect} \)
(5) USE DEF \( \text{Pcri} \)
(5) USE DEF \( \text{Pcrx} \)
(5) USE \( \text{ThmPcrInitIsPcr} \)
(5) USE \( \text{ThmPcrExtendIsPcr} \)
(5) \( \text{semPcr} = \text{SemProtect} \) BY DEF \( \text{InInSemProtect}, \text{InSem} \)
(5) USE \( \text{AssSemHappy} \)
(5) QED BY \( \text{ThmPcrExtendAnticollision} \)
LogInApp

LogInTs

CheckTsIsCurrent

QED BY DEF LogInTs

LogInNs

InvOneLog

InvVerifiableRevocation

QED BY DEF InvOneLog

InvVerifiableRevocation

InvOneLog

InvVerifiableRevocation

InvOneLog

QED BY DEF InvOneLog

CurrentTsLog

Clearing nv current erases the log from the nv ram.

BugRecovNoClrCur \triangleq \text{FALSE}

QED BY DEF InvType, Nv

LogInApp

LogInTs

InvOneLog

InvOneLog

InvVerifiableRevocation

QED BY DEF InvVerifiableRevocation, InvOneLog

NextSemRecov3

(2) 13. CASE NextSemRecov3

(3) USE NextSemRecov3

(3) USE DEF NextSemRecov3

(3) UNCHANGED CheckTsIsCurrent(chkptts) BY DEF CheckTsIsCurrent

(3) UNCHANGED AllCurrentTs

(4) USE DEF AllCurrentTs

(4) USE DEF CheckTsIsCurrent

(4) QED BY DEF LogInTs
Extending sem pcr to SemiHappy puts the log in the application pcr, provided that the seal pcr contains SealReboot.

But in the current state the log has no domicile. So the fact that its domicile might be the application pcr in the next state does not require a proof that it is not living anywhere else, since we get that for free.

Cancels SealReboot if we had it, which erases any log that had been in the application pcr.

This depends on BugSealNoExt $\triangleq$ FALSE

If the log was not in the application pcr, the resulting seal attestation will not be valid, so there is no change in AllCurrentTs or LogSummaryInTs.
\( \langle 5 \rangle \) PICK \( ts \in SignedTs : ts = NextSealTs! : ts \) BY DEF InvType, SignedTs
\( \langle 5 \rangle \) \( \forall ts1 \in tsvalues : CheckTsIsCurrent(ts1)' \Rightarrow ts1 \in tsvalues \)
\( \langle 6 \rangle \) \( tsvalues' = tsvalues \cup \{ ts \} \) OBVIOUS
\( \langle 6 \rangle \) \( \neg CheckTsIsCurrent(ts)' \) BY DEF LogInApp
\( \langle 6 \rangle \) QED OBVIOUS
\( \langle 5 \rangle \) QED OBVIOUS
\( \langle 4 \rangle \) UNCHANGED LogInTs \hspace{1cm} \text{BY DEF LogInTs}
\( \langle 4 \rangle \) UNCHANGED CurrentTsLog \hspace{1cm} \text{BY DEF CurrentTsLog}
\( \langle 4 \rangle \) InvOneLog! goal! obtains’ \hspace{1cm} \text{BY DEF IsOnLog, InvOneLog}
\( \langle 4 \rangle \) InvOneLog! goal! revokes’ \hspace{1cm} \text{BY DEF IsOnLog, InvOneLog}
\( \langle 4 \rangle \) QED BY DEF InvOneLog

If the log was in the application pcr, the resulting seal attestation will be valid. But then the old AllCurrentTs had to be empty, since the log could not have been in the seal attestations.

\( \langle 3 \rangle \) CASE LogInApp

Use PICK to make \( ts \) a CONSTANT so that \( CheckTsIsCurrent(ts)' \) means the \( ts \) picked now evaluated with \( CheckTsIsCurrent \) in the next state.

\( \langle 4 \rangle \) PICK \( ts \in SignedTs : ts = NextSealTs! : ts \) BY DEF InvType, SignedTs
\( \langle 4 \rangle \) CheckTsIsCurrent(ts)'
\( \langle 5 \rangle \) CheckTsIsCurrent(ts) BY DEF CheckTsIsCurrent, LogInApp
\( \langle 5 \rangle \) QED BY DEF CheckTsIsCurrent
\( \langle 4 \rangle \) AllCurrentTs' = \{ ts \}
\( \langle 5 \rangle \) \( \forall ts1 \in tsvalues \cup \{ chkptts \} : \neg CheckTsIsCurrent(ts1)' \)
\( \langle 6 \rangle \) AllCurrentTs = \{ \}
\( \langle 7 \rangle \) \( \neg LogInTs \) BY DEF InvOneLog
\( \langle 7 \rangle \) QED BY DEF LogInTs
\( \langle 6 \rangle \) \( \neg CheckTsIsCurrent(chkptts)' \)
\( \langle 7 \rangle \) \( \neg CheckTsIsCurrent(chkptts) \) BY DEF AllCurrentTs
\( \langle 7 \rangle \) QED BY DEF CheckTsIsCurrent
\( \langle 6 \rangle \) QED BY DEF CheckTsIsCurrent
\( \langle 5 \rangle \) tsvalues' = tsvalues \cup \{ ts \} \) OBVIOUS
\( \langle 5 \rangle \) ts \in AllCurrentTs' BY DEF AllCurrentTs
\( \langle 5 \rangle \) QED BY DEF AllCurrentTs
\( \langle 4 \rangle \) ts1, ts2 \in AllCurrentTs' : ts1.appPcr = ts2.appPcr \) OBVIOUS
\( \langle 4 \rangle \) LogInTs' \hspace{1cm} \text{BY DEF LogInTs}
\( \langle 4 \rangle \) CurrentTsLog' = ts.appPcr \hspace{1cm} \text{BY DEF CurrentTsLog}
\( \langle 4 \rangle \) InvOneLog! goal! obtains' \hspace{1cm} \text{BY DEF IsOnLog, InvOneLog}
\( \langle 4 \rangle \) InvOneLog! goal! revokes' \hspace{1cm} \text{BY DEF IsOnLog, InvOneLog}
\( \langle 4 \rangle \) QED BY DEF InvOneLog
\( \langle 3 \rangle \) QED OBVIOUS
(2) 15. CASE NextEnterSemChkpt
   (3) USE NextEnterSemChkpt
   (3) USE DEF NextEnterSemChkpt
   (3) USE DEF InSem
   (3) UNCHANGED LogInNv  BY DEF LogInNv
Cancels SemHappy if we had it, so erases any log that might have been in the application pcr.

(3) ¬LogInApp'
   (4) semPer' ≠ SemHappy
   (5) USE DEF SemHappy
   (5) USE DEF SemProtect
   (5) USE DEF Pcri
   (5) USE DEF Pcrx
   (5) USE ThmPcrInitIsPcr
   (5) USE ThmPcrExtendIsPcr
   (5) PcrLeq(SemProtect, SemHappy) BY ThmPcrExtendLeq
   (5) ¬PcrLeq(SemHappy, SemProtect) BY ThmPcrExtendSelfUnreachable
   (5) QED BY ThmPcrLeqIsAntisymmetric
(4) QED BY DEF LogInApp
Overwrites chkptts with a value from tsvalues, so if chkptts had been the only seal log, we just erased it.

(3) LogInTs' ⇒ LogInTs
   (4) HAVE LogInTs'
   (4) AllCurrentTs' ≠ {} BY DEF LogInTs
   (4) USE DEF AllCurrentTs
   (4) USE DEF CheckTsIsCurrent
   (4) chkptts' ∈ tsvalues' OBVIOUS
   (4) ∃ ts ∈ tsvalues' : CheckTsIsCurrent(ts) OBVIOUS
(4) QED BY DEF LogInTs
Any remaining current seal attestations existed previously, so they must contain the same log.

(3) ∀ ts1, ts2 ∈ AllCurrentTs' : ts1.appPcr = ts2.appPcr
   (4) TAKE ts1, ts2 ∈ AllCurrentTs'
   (4) USE DEF AllCurrentTs
   (4) USE DEF CheckTsIsCurrent
   (4) chkptts' ∈ tsvalues' OBVIOUS
   (4) ts1 ∈ AllCurrentTs' OBVIOUS
   (4) ts2 ∈ AllCurrentTs' OBVIOUS
(4) QED BY DEF InvOneLog
If there are any remaining current seal attestations, the log in them has to be the same as before.

(3) LogInTs' ⇒ UNCHANGED CurrentTsLog
   (4) HAVE LogInTs'
   (4) ∃ ts ∈ AllCurrentTs' : ts ∈ AllCurrentTs
   (5) USE DEF LogInTs
   (5) USE DEF AllCurrentTs
   (5) USE DEF CheckTsIsCurrent
   (5) ∀ ts ∈ AllCurrentTs' : ts ∈ AllCurrentTs' OBVIOUS
\langle 5 \rangle \text{ QED OBVIOUS}
\langle 4 \rangle \forall t_1 \in \text{AllCurrentTs}':
\forall t_0 \in \text{AllCurrentTs}:
\text{ts}_1.\text{appPcr} = \text{ts}_0.\text{appPcr}
\text{BY DEF InvOneLog}
\langle 4 \rangle \text{ QED BY DEF CurrentTsLog}
\langle 3 \rangle \text{ InvOneLog}!\text{goal'}\text{obtains'} \text{BY DEF IsOnLog, InvOneLog}
\langle 3 \rangle \text{ InvOneLog}!\text{goal'}\text{revokes'} \text{BY DEF IsOnLog, InvOneLog}
\langle 3 \rangle \text{ InvVerifiableRevocation'} \text{BY DEF InvVerifiableRevocation, InvOneLog}
\langle 3 \rangle \text{ QED BY DEF InvOneLog}

\text{NextSemChkpt1WhenCorrect}
\langle 2 \rangle 16. \text{ CASE NextSemChkpt1WhenCorrect}
\langle 3 \rangle \text{ USE NextSemChkpt1WhenCorrect}
\langle 3 \rangle \text{ USE DEF NextSemChkpt1WhenCorrect}
\langle 3 \rangle \text{ UNCHANGED CheckTsIsCurrent(chkptts)BY DEF CheckTsIsCurrent}
\langle 3 \rangle \text{ UNCHANGED AllCurrentTs}
\langle 4 \rangle \text{ USE DEF AllCurrentTs}
\langle 4 \rangle \text{ USE DEF CheckTsIsCurrent}
\langle 4 \rangle \text{ QED BY DEF LogInTs}
\langle 3 \rangle \text{ UNCHANGED CurrentTsLogBY DEF CurrentTsLog}
\langle 3 \rangle \text{ UNCHANGED LogInNu BY DEF LogInNu}
\langle 3 \rangle \text{ UNCHANGED LogInApp BY DEF LogInApp}
\langle 3 \rangle \text{ UNCHANGED LogInTs BY DEF LogInTs}

\text{EnterSemChkptPredicate} \text{guarantees that the log is in the seal attestations (in particular, in chkptts).}

This depends on
\land \text{BugChkptNoCheckTsHappy} \triangleq \text{FALSE}
\land \text{BugChkptNoCheckTsSeal} \triangleq \text{FALSE}
\land \text{BugChkptNoCheckTsCtr} \equiv \text{FALSE}
\langle 3 \rangle \text{ LogInTs \land CheckTsIsCurrent(chkptts) \land CurrentTsLog = chkptts.\text{appPcr}}
\langle 4 \rangle \text{ USE DEF EnterSemChkptPredicate}
\langle 4 \rangle \text{ BugChkptNoCheckTsHappy = FALSEBY DEF BugChkptNoCheckTsHappy}
\langle 4 \rangle \text{ BugChkptNoCheckTsSeal = FALSEBY DEF BugChkptNoCheckTsSeal}
\langle 4 \rangle \text{ BugChkptNoCheckTsCtr = FALSEBY DEF BugChkptNoCheckTsCtr}
\langle 4 \rangle \text{ CheckTsIsCurrent(chkptts)BY DEF CheckTsIsCurrent}
\langle 4 \rangle \text{ AllCurrentTs} \neq \emptyset \text{BY DEF AllCurrentTs}
\langle 4 \rangle \text{ CurrentTsLog = chkptts.\text{appPcr}}
\langle 5 \rangle \forall t_0 \in \text{AllCurrentTs}: t_0.\text{appPcr} = \text{chkptts.\text{appPcr}}
\text{BY DEF AllCurrentTs, InvOneLog}
\langle 5 \rangle \text{ QED BY DEF CurrentTsLog}
\langle 4 \rangle \text{ QED BY DEF LogInTs}
\langle 3 \rangle \text{ InvOneLog}!\text{goal'}\text{obtains'} \text{BY DEF IsOnLog, InvOneLog}
\langle 3 \rangle \text{ InvOneLog}!\text{goal'}\text{revokes'} \text{BY DEF IsOnLog, InvOneLog}
\langle 3 \rangle \text{ InvVerifiableRevocation'} \text{BY DEF InvVerifiableRevocation, InvOneLog}
NextSemChkpt1 WhenIncorrect

(2) 17. CASE NextSemChkpt1 WhenIncorrect
    (3) USE NextSemChkpt1 WhenIncorrect
    (3) USE DEF NextSemChkpt1 WhenIncorrect
    (3) UNCHANGED CheckTsIsCurrent(chkpts) BY DEF CheckTsIsCurrent
    (3) UNCHANGED AllCurrentTs
        (4) USE DEF AllCurrentTs
        (4) USE DEF CheckTsIsCurrent
        (4) QED BY DEF LogInTs
    (3) UNCHANGED CurrentTsLog BY DEF CurrentTsLog
    (3) UNCHANGED LogInNs
        BY DEF LogInNs

Extending sem pcr with Unhappy results in something other than SemHappy, which indicates that the log is not in the application pcr.

(3) ¬LogInApp′
    (4) semPcr′ ≠ SemHappy
        (5) USE DEF SemHappy
        (5) USE DEF SemProtect
        (5) USE DEF Pcri
        (5) USE DEF Pcrx
        (5) USE ThmPerInitIsPcr
        (5) USE ThmPerExtendIsPcr
        (5) semPcr = SemProtect BY DEF InvInSemProtect, InSem
        (5) USE AssSemHappy
        (5) QED BY ThmPerExtendAnticollision
    (4) QED BY DEF LogInApp
(3) UNCHANGED LogInTs
    (4) USE DEF AllCurrentTs
    (4) USE DEF CheckTsIsCurrent
    (4) QED BY DEF LogInTs
(3) InvOneLog′! goal! obtains′ BY DEF IsOnLog, InvOneLog
(3) InvOneLog′! goal! revokes′ BY DEF IsOnLog, InvOneLog
(3) InvVerifiableRevocation′ BY DEF InvVerifiableRevocation, InvOneLog
(3) QED BY DEF InvOneLog

NextSemChkpt2

(2) 18. CASE NextSemChkpt2
    (3) USE NextSemChkpt2
    (3) USE DEF NextSemChkpt2
    (3) UNCHANGED CheckTsIsCurrent(chkpts) BY DEF CheckTsIsCurrent
    (3) UNCHANGED AllCurrentTs
        (4) USE DEF AllCurrentTs
        (4) USE DEF CheckTsIsCurrent
C PROOF OF INITIAL PASTURE

(4) QED BY DEF LogInTs
(3) UNCHANGED CurrentTsLog BY DEF CurrentTsLog
(3) UNCHANGED LogInNu
  (4) USE DEF InvType
  (4) USE DEF Nu
  (4) QED BY DEF LogInNu
(3) UNCHANGED LogInApp BY DEF LogInApp
(3) UNCHANGED LogInTs BY DEF LogInTs

Storing the log from chkptts to the nu ram.

This depends on BugChkptSaveCurApp \( \triangleq \) false

(3) nu'.appPcr = chkptts.appPcr
  (4) BugChkptSaveCurApp = FALSE BY DEF BugChkptSaveCurApp
  (4) QED BY DEF InvType, Nu
(3) InvOneLog! goal! obtains' BY DEF IsOnLog, InvOneLog
(3) InvOneLog! goal! revokes' BY DEF IsOnLog, InvOneLog
(3) InvVerifiableRevocation' BY DEF InvVerifiableRevocation, InvOneLog
(3) QED BY DEF InvOneLog

NextSemChkpt3

(2) 19. CASE NextSemChkpt3
  (3) USE NextSemChkpt3
  (3) USE DEF NextSemChkpt3
  (3) UNCHANGED LogInNu BY DEF LogInNu
  (3) UNCHANGED LogInApp BY DEF LogInApp

Since no signed ts seal can have a bootCtr greater than the current bootCtr, incrementing bootCtr erases any log that might have been in a seal attestation.

This depends on BugChkptNoIncCtr \( \triangleq \) false

(3) ¬LogInTs'
  (4) USE DEF AllCurrentTs
  (4) USE DEF CheckTsIsCurrent
  (4) \( \forall ts \in tsvalues' \cup \{chkptts'\} : ts \in SignedTs \Rightarrow ts.bootCtr \neq bootC' \)
    (5) TAKE ts \in tsvalues' \cup \{chkptts'\}
    (5) ts \in tsvalues' \cup \{chkptts\} OBVIOUS
    (5) HAVE ts \in SignedTs
    (5) ts.bootCtr \leq bootC' BY DEF InvSignedTsLeqBoot
    (5) ts.bootC' \in Nat BY DEF SignedTs
    (5) bootC' \in Nat BY DEF InvType
    (5) bootC' < bootC'
      (6) BugChkptNoIncCtr = FALSE BY DEF BugChkptNoIncCtr
      (6) QED BY ThmNatInc
    (5) ts.bootC' < bootC' BY ThmNatLeqLt
    (5) QED BY ThmNatLeqXorGt, ThmNatLeqIsReflexive
(4) QED BY DEF LogInTs
Erasas all current ts seal attestations.

(3) AllCurrentTs' = {} ∧ ¬CheckTsIsCurrent(chkptts)'
(4) USE DEF AllCurrentTs
(4) USE DEF CheckTsIsCurrent
(4) QED BY DEF LogInTs

(3) InvOneLog!goal obtains' BY DEF IsOnLog, InvOneLog
(3) InvOneLog!goal revokes' BY DEF IsOnLog, InvOneLog
(3) InvVerifiableRevocation' BY DEF InvVerifiableRevocation, InvOneLog
(3) QED BY DEF InvOneLog

NextSemChkpt4

(2) 20. CASE NextSemChkpt4
(3) USE NextSemChkpt4
(3) USE DEF NextSemChkpt4
(3) UNCHANGED CheckTsIsCurrent(chkptts) BY DEF CheckTsIsCurrent
(3) UNCHANGED AllCurrentTs
(4) USE DEF AllCurrentTs
(4) USE DEF CheckTsIsCurrent
(4) QED BY DEF LogInTs
(3) UNCHANGED CurrentTsLog BY DEF CurrentTsLog

Setting nv.current indicates that the log is in the nv ram.
This depends on BugChkptNoSetCur ∆ = FALSE

(3) LogInNv'
(4) nv'.current
  (5) BugChkptNoSetCur = FALSE BY DEF BugChkptNoSetCur
  (5) QED BY DEF InvType, Nv
(4) QED BY DEF LogInNv
(3) UNCHANGED LogInApp BY DEF LogInApp
(3) UNCHANGED LogInTs BY DEF LogInTs

Since nv.current was changed, the prover needs to see the type of nv to know that the appPcr field did not change.

(3) UNCHANGED nv.appPcr BY DEF InvType, Nv
(3) InvOneLog!goal obtains' BY DEF IsOnLog, InvOneLog
(3) InvOneLog!goal revokes' BY DEF IsOnLog, InvOneLog
(3) InvVerifiableRevocation' BY DEF InvVerifiableRevocation, InvOneLog
(3) QED BY DEF InvOneLog

NextSemChkpt5

(2) 21. CASE NextSemChkpt5
(3) USE NextSemChkpt5
(3) USE DEF NextSemChkpt5
(3) UNCHANGED CheckTsIsCurrent(chkptts) BY DEF CheckTsIsCurrent
Extending sem pcr with Unhappy results in something other than \( \text{SemHappy} \), which indicates that the log is not in the application pcr.

\[
\text{By Def} \quad \text{LogInApp}
\]

It is an invariant of the specification.

**Theorem** \( \text{ThmInvOneLog} \) \( \triangleq \)

\[ \text{Spec} \Rightarrow \square \text{InvOneLog} \]

**Proof**

\( \text{Init} \Rightarrow \text{InvOneLog} \) \( \text{by} \ \text{ThmInitInvOneLog} \)

\( \text{InvOneLog} \land [\text{Next}]_{\text{vars}} \Rightarrow \text{InvOneLog}' \) \( \text{by} \ \text{ThmNextInvOneLog} \)

\( \text{QED} \ \text{by} \ \text{RuleINV} \)
PROOF OF INVARIANT \textit{InvAccessUndeniability}

It is an invariant of the specification.

\textbf{THEOREM} \textit{ThmInvAccessUndeniability} \(\triangleq\)
\begin{align*}
\text{Spec} &\Rightarrow \Box \text{InvAccessUndeniability} \\
\text{PROOF} & \\
\langle 1 \rangle & \text{InvOneLog} \Rightarrow \text{InvAccessUndeniability} \\
\langle 2 \rangle & \text{HAVE InvOneLog} \\
\langle 3 \rangle & \text{USE DEF InvOneLog} \\
\langle 4 \rangle & \text{USE DEF InvAccessUndeniability} \\
\langle 5 \rangle & \text{BugAuditNoCheckHappy} = \text{FALSE} \text{BY DEF BugAuditNoCheckHappy} \\
\langle 6 \rangle & \text{BugAuditNoCheckSeal} = \text{FALSE} \text{BY DEF BugAuditNoCheckSeal} \\
\langle 7 \rangle & \text{CASE} \neg \text{LogInApp} \text{ BY DEF LogInApp \text{unable to audit}} \\
\langle 8 \rangle & \text{CASE} \text{LogInApp} \text{ BY DEF IsOnLog \text{audit}} \\
\langle 9 \rangle & \Box \text{QED OBVIOUS} \\
\langle 1 \rangle & \text{Spec} \Rightarrow \Box \text{InvOneLog} \text{BY ThmInvOneLog} \\
\langle 1 \rangle & \text{QED}
\end{align*}

PROOF OF INVARIANT \textit{InvVerifiableRevocation}

It is an invariant of the specification.

\textbf{THEOREM} \textit{ThmInvVerifiableRevocation} \(\triangleq\)
\begin{align*}
\text{Spec} &\Rightarrow \Box \text{InvVerifiableRevocation} \\
\text{PROOF} & \\
\langle 1 \rangle & \text{InvOneLog} \Rightarrow \text{InvVerifiableRevocation} \\
\langle 2 \rangle & \text{HAVE InvOneLog} \\
\langle 3 \rangle & \text{USE DEF InvOneLog} \\
\langle 4 \rangle & \Box \text{QED OBVIOUS} \\
\langle 1 \rangle & \text{Spec} \Rightarrow \Box \text{InvOneLog} \text{BY ThmInvOneLog}
\end{align*}
\[1\] \text{QED}
D Specification of Optimized Pasture

MODULE OptPastureNode

EXTENDS Naturals, Sequences, FiniteSets

Override one of the following definitions to introduce a bug in the specification.

\[
\begin{align*}
\text{BugObtainAccessNoCheckHappy} & \triangleq \text{FALSE} \\
\text{BugProveRevokeNoCheckHappy} & \triangleq \text{FALSE} \\
\text{BugReconNoCheckApp} & \triangleq \text{FALSE} \\
\text{BugReconNoCheckCur} & \triangleq \text{FALSE} \\
\text{BugReconNoClrCur} & \triangleq \text{FALSE} \\
\text{BugSealNoExt} & \triangleq \text{FALSE} \\
\text{BugChkptNoCheckTsHappy} & \triangleq \text{FALSE} \\
\text{BugChkptNoCheckTsCtr} & \triangleq \text{FALSE} \\
\text{BugChkptSaveCurApp} & \triangleq \text{FALSE} \\
\text{BugChkptNoIncCtr} & \triangleq \text{FALSE} \\
\text{BugChkptNoSetCur} & \triangleq \text{FALSE} \\
\text{BugAuditNoCheckHappy} & \triangleq \text{FALSE}
\end{align*}
\]

\text{liveness bug; not actually a safety bug}

\text{PCR INITIALIZATION VALUES}

\text{CONSTANT PcriAPPBOOT} \quad \text{reboot initialization of app pcr}
\text{CONSTANT PcriSEMBOOT} \quad \text{reboot initialization of sem pcr}
\text{CONSTANT PcriSEMPROTECT} \quad \text{secure execution mode entry of sem pcr}

\text{Pcri} \triangleq \{ \\
PcriAPPBOOT, \\
PcriSEMBOOT, \\
PcriSEMPROTECT \\
\}

\text{Initialization of sem pcr via boot and via secure execution mode entry must be different.}

\text{ASSUME AssSemProtect} \triangleq \text{PcriSEMBOOT} \neq \text{PcriSEMPROTECT}

\text{PCR EXTENSION VALUES}

\text{CONSTANT PcrxHAPPY} \quad \text{recover is happy}
CONSTANT $P_{crx\text{-}UNHAPPY}$ recover is unhappy or checkpoint is unhappy/finished
CONSTANT $P_{crx\text{-SEAL}}$ seal marker
CONSTANT $P_{crx\text{-OBTAIN}}$ obtain access operation
CONSTANT $P_{crx\text{-REVOKE}}$ revoke access operation

$P_{crx} \triangleq$

\[
\begin{cases}
P_{crx\text{HAPPY}}, \\
P_{crx\text{UNHAPPY}}, \\
P_{crx\text{SEAL}}, \\
P_{crx\text{OBTAIN}}, \\
P_{crx\text{REVOKE}}
\end{cases}
\]

Extension for obtain access and extension for revoke access must be different.

ASSUME $\text{AssObtainNeqRevoke} \triangleq P_{crx\text{OBTAIN}} \neq P_{crx\text{REVOKE}}$

Extension for happy and extension for unhappy must be different.

ASSUME $\text{AssSemHappy} \triangleq P_{crx\text{HAPPY}} \neq P_{crx\text{UNHAPPY}}$

---

**PCR VALUES**

A pcr value is modeled as an initialization followed by a sequence of extensions.

$P_{cr} \triangleq$

\[
[\quad \text{init} : P_{cri}, \quad \text{extq} : \text{Seq}(P_{crx}) \quad ]
\]

Initial pcr value.

$P_{cr\text{Init}}(i) \triangleq$

\[
[\quad \text{init} \mapsto i, \quad \text{extq} \mapsto \langle \rangle \quad ]
\]

Extend a pcr value.

$P_{cr\text{Extend}}(p, x) \triangleq$
\[
\begin{align*}
\text{init} &\mapsto p.\text{init}, \\
\text{extq} &\mapsto \text{Append}(p.\text{extq}, x)
\end{align*}
\]

Number of extensions in a pcr value.

\[PcrLen(p) \triangleq \text{Len}(p.\text{extq})\]

\(Pcr\ s \leq Pcr\ t\). This means that with zero or more extensions, you can extend \(s\) to reach \(t\). This is a partial order relation.

\[PcrLeq(s, t) \triangleq \]
LET
\[
\begin{align*}
\text{sinit} &\triangleq s.\text{init} \\
\text{sextq} &\triangleq s.\text{extq} \\
\text{sn} &\triangleq \text{Len}(\text{sextq}) \\
\text{tinit} &\triangleq t.\text{init} \\
\text{textq} &\triangleq t.\text{extq} \\
\text{tn} &\triangleq \text{Len}(\text{textq}) \\
\text{uextq} &\triangleq \text{SubSeq}(\text{textq}, 1, \text{sn})
\end{align*}
\]
IN
\[
\begin{align*}
\land \text{sinit} = \text{tinit} \\
\land \text{sn} \leq \text{tn} \\
\land \text{sextq} = \text{uextq}
\end{align*}
\]

Determine if a pcr value has been extended.

\[PcrHasExtension(p) \triangleq \]
\[PcrLen(p) > 0\]

Assuming a pcr value has been extended, get the prior pcr value that this one was extended from. We assume the adversary can compute this by watching all pcr computations.

\[PcrPrior(p) \triangleq \]
CASE \(PcrHasExtension(p) \rightarrow \)
LET
\[
\begin{align*}
n &\triangleq \text{Len}(p.\text{extq}) - 1
\end{align*}
\]
IN
\[
\begin{align*}
\text{init} &\mapsto p.\text{init}, \\
\text{extq} &\mapsto \text{SubSeq}(p.\text{extq}, 1, n)
\end{align*}
\]

Assuming a pcr value has been extended, get the last extension. We assume the adversary can compute this by watching all pcr computations.
\[ PcrLastExtension(p) \triangleq \]
\[
\begin{cases}
\text{CASE } PcrHasExtension(p) \rightarrow \\
p\text{ext}[\text{Len}(p\text{ext})]
\end{cases}
\]

**WELL KNOWN PCR VALUES**

Value of the application pcr attained by rebooting.
\[ AppReboot \triangleq PcrInit(PcriAPPBOOT) \]

Value of the secure execution mode pcr attained by rebooting.
\[ SemReboot \triangleq PcrInit(PcriSEMBOOT) \]

Value of the secure execution mode pcr attained by entering the protected module in secure execution mode. This value permits access to the Pasture protected Nv ram.
\[ SemProtect \triangleq PcrInit(PcriSEMPROTECT) \]

Value of the secure execution mode pcr that indicates that Pasture is happy. Recovery has been properly performed and bound keys may be used. Checkpoint has not yet been invoked.
\[ SemHappy \triangleq PcrExtend(SemProtect, PcrxHAPPY) \]

**PC VALUES**

- anywhere not in secure execution mode
\[ PcIDLE \triangleq \text{"idle"} \]

- steps in secure execution mode within recover
\[ PcRECOV1 \triangleq \text{"reco1"} \]
\[ PcRECOV2 \triangleq \text{"reco2"} \]
\[ PcRECOV3 \triangleq \text{"reco3"} \]
\[ PcRecov \triangleq \{PcRECOV1, PcRECOV2, PcRECOV3\} \]

- steps in secure execution mode within checkpoint
\[ PcCHKPT1 \triangleq \text{"chkpt1"} \]
\[ PcCHKPT2 \triangleq \text{"chkpt2"} \]
\[ PcCHKPT3 \triangleq \text{"chkpt3"} \]
\[ PcCHKPT4 \triangleq \text{"chkpt4"} \]
\[ PcCHKPT5 \triangleq \text{"chkpt5"} \]
\[ PcChkpt \triangleq \{PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5\} \]
\[ P_c \triangleq \{PcIDLE\} \cup PcRecov \cup PcChkp \]

**PROTECTED NV RAM STATE**

\[ N_v \triangleq \left[ \begin{array}{l}
appPcr : Pcr, \text{ copy of the application pcr} \\
current : BOOLEAN \quad \text{copy of application pcr is current}
\end{array} \right] \]

\[ InitNv \triangleq \left[ \begin{array}{l}
appPcr \mapsto \text{AppReboot}, \\
current \mapsto \text{TRUE}
\end{array} \right] \]

**SEAL OPERATION TRANSPORT SESSION STATE**

We model the signed “seal operation” transport session as a record of the input values required in order for the transport session \(TPM\) signature to be valid.

\[ SignedTs \triangleq \left[ \begin{array}{l}
semPcr : Pcr, \text{ copy of the secure execution mode pcr on entry} \\
appPcr : Pcr, \text{ copy of the application pcr on entry} \\
bootCt : \text{Nat} \text{ copy of the reboot counter on entry}
\end{array} \right] \]

The adversary cannot forge a correctly signed seal attestation. We model all incorrectly signed ones as the following single value.

\[ NullTs \triangleq \text{CHOSE NullTs : NullTs} \notin SignedTs \]

\[ Ts \triangleq SignedTs \cup \{NullTs\} \]
STATE

VARIABLE $nv$ Pasture’s protected NV RAM region
VARIABLE $appPcr$ the application pcr
VARIABLE $semPcr$ the secure execution mode pcr
VARIABLE $bootCtr$ the reboot counter
VARIABLE $pc$ the pc
VARIABLE $chkptts$ ts passed to sem within checkpoint
VARIABLE $tsvalues$ what ts values are known
VARIABLE $obtains$ decisions to obtain access
VARIABLE $revokes$ decisions to prove revoke access

Tuple of all variables.

$vars \triangleq (nv, appPcr, semPcr, bootCtr, pc, chkptts, tsvalues, obtains, revokes)$

STATE PREDICATES

The node is currently in secure execution mode.

$InSem \triangleq pc \neq PcIDLE$

NEXT STATE RELATION

Employ a key binding to obtain access to read a message.

If the last extension to the application pcr was an OBTAIN operation, then in full generality there could have been a key bound to this application pcr value. So record the information that we obtained access to this key binding.

$NextObtainAccess \triangleq$

LET

$pcr0 \triangleq appPcr$ current app pcr
$pcr0 \triangleq PcrPrior(pcr1o)$ prior app pcr
$x \triangleq PcrLastExtension(pcr1o)$ presumed OBTAIN extension

IN

$\land \neg InSem$ must not be in secure execution mode
∧ PcrHasExtension(pcr1₀)  
∧ x = PcrxOBTAIN  
∧ x₀ = PcrxREVOKE  
∧ PcrLastExtension(pcr₁₀)  
∧ PcrPrior(pcr₁₀)  
∧ PcrExtend(pcr₀, x)  
∧ ¬InSem  
∧ PcrHasExtension(pcr₁₀)  
∧ x = PcrxREVOKE  
∧ InSem 
∧ PcrHasExtension(pcr₁₀) 
∧ x = PcrxREVOKE 
∧ ¬InSem 
∧ PcrHasExtension(pcr₁₀) 
∧ x = PcrxREVOKE 
∧ ¬InSem
\[\begin{align*}
\land \text{UNCHANGED } \text{bootCtr} \\
\land \text{UNCHANGED } \text{pc} \\
\land \text{UNCHANGED } \text{chkptts} \\
\land \text{UNCHANGED } \text{tsvalues} \\
\land \text{UNCHANGED } \text{obtains}
\end{align*}\]

Reboot the node.

This can happen at absolutely any time, due to adversarial action. However, if it happens without going through the proper seal and checkpoint actions, liveness may be lost.

Resetting \text{chkptts} to its initial value erases information and thus reduces the number of distinct states that model checking has to explore. But note that the adversary could always remember whatever value \text{chkptts} had before and call \text{sem checkpoint} with that value.

\text{NextReboot} \triangleq \\
\land \text{appPcr'} = \text{AppReboot} \\
\land \text{semPcr'} = \text{SemReboot} \\
\land \text{pc'} = \text{PcIDLE} \\
\land \text{chkptts'} = \text{NullTs} \\
\land \text{UNCHANGED } \text{nv} \\
\land \text{UNCHANGED } \text{bootCtr} \\
\land \text{UNCHANGED } \text{tsvalues} \\
\land \text{UNCHANGED } \text{obtains} \\
\land \text{UNCHANGED } \text{revokes}

Forget one of the seal transport sessions.

This can happen at absolutely any time, and represents a loss of knowledge by the adversary which enables additional execution paths to fall within the model checking constraints.

\text{NextForgetSealTs} \triangleq \\
\exists t \in \text{tsvalues} : \\
\land \text{tsvalues'} = \text{tsvalues} \setminus \{t\} \\
\land \text{UNCHANGED } \text{nv} \\
\land \text{UNCHANGED } \text{appPcr} \\
\land \text{UNCHANGED } \text{semPcr} \\
\land \text{UNCHANGED } \text{bootCtr} \\
\land \text{UNCHANGED } \text{pc} \\
\land \text{UNCHANGED } \text{chkptts} \\
\land \text{UNCHANGED } \text{obtains} \\
\land \text{UNCHANGED } \text{revokes}
Extend application pcr arbitrarily.  
In proper execution, this action is performed as necessary after reboot to re-extend the application pcr to its last checkpoint value.  
In proper execution, this action is performed as desired to decide upon reading or deleting messages.  
The adversary can perform this action at any idle time.

\[
\text{NextExtendAppPcr} \triangleq \\
\land \lnot \text{InSem} \\
\land \exists x \in Pcrx : \\
\land appPcr' = Pcr\text{Extend}(appPcr, x) \\
\land \text{UNCHANGED } nv \\
\land \text{UNCHANGED } semPcr \\
\land \text{UNCHANGED } bootCtr \\
\land \text{UNCHANGED } pc \\
\land \text{UNCHANGED } chkptts \\
\land \text{UNCHANGED } tsvalues \\
\land \text{UNCHANGED } obtains \\
\land \text{UNCHANGED } revokes \\
\]

Extend secure execution mode pcr arbitrarily, due to adversarial action.

\[
\text{NextExtendSemPcr} \triangleq \\
\land \lnot \text{InSem} \quad \text{must not be in secure execution mode} \\
\land \exists x \in Pcrx : \\
\land semPcr' = Pcr\text{Extend}(semPcr, x) \\
\land \text{UNCHANGED } nv \\
\land \text{UNCHANGED } appPcr \\
\land \text{UNCHANGED } bootCtr \\
\land \text{UNCHANGED } pc \\
\land \text{UNCHANGED } chkptts \\
\land \text{UNCHANGED } tsvalues \\
\land \text{UNCHANGED } obtains \\
\land \text{UNCHANGED } revokes \\
\]

Increment reboot counter arbitrarily, due to adversarial action.

\[
\text{NextIncBootCtr} \triangleq \\
\land \lnot \text{InSem} \quad \text{must not be in secure execution mode} \\
\land bootCtr' = bootCtr + 1 \\
\land \text{UNCHANGED } nv \\
\land \text{UNCHANGED } appPcr \\
\land \text{UNCHANGED } semPcr \\
\land \text{UNCHANGED } pc \\
\land \text{UNCHANGED } chkptts \\
\land \text{UNCHANGED } tsvalues \\
\]
Enter secure execution mode within recovery.

In proper execution, this action is performed during system boot after the application pcr has been re-extended to its last checkpoint value. This re-extension is performed by untrusted code that reads the necessary extension values from a stable log.

The adversary can perform this action at any idle time. But it will not do any good unless the application pcr contains the last checkpoint value and the last checkpoint value is marked as current.

\[
\text{NextEnterSemRecov} \triangleq \\
\begin{align*}
\& \neg \text{InSem} \\
\& \text{semPcr} = \text{SemProtect} \\
\& \text{pc} = \text{PcRECOV1} \\
\& \text{UNCHANGED } \text{nv} \\
\& \text{UNCHANGED } \text{appPcr} \\
\& \text{UNCHANGED } \text{bootCtr} \\
\& \text{UNCHANGED } \text{chkptts} \\
\& \text{UNCHANGED } \text{tsvalues} \\
\& \text{UNCHANGED } \text{obtains} \\
\& \text{UNCHANGED } \text{revokes}
\end{align*}
\]

Predicate for correct entry to secure execution mode within recovery.

\[
\text{EnterSemRecovPredicate} \triangleq \\
\begin{align*}
\& \text{IF BugRecovNoCheckApp THEN TRUE ELSE} \\
\& \text{EN} \text{v.appPcr} = \text{appPcr} \\
\& \text{IF BugRecovNoCheckCur THEN TRUE ELSE} \\
\& \text{EN} \text{v.current}
\end{align*}
\]

Secure execution mode within recovery step 1, when there is correct entry.

\[
\text{NextSemRecov1 WhenCorrect} \triangleq \\
\begin{align*}
\& \text{pc} = \text{PcRECOV1} \\
\& \text{EnterSemRecovPredicate} \\
\& \text{pc} = \text{PcRECOV2} \\
\& \text{UNCHANGED } \text{nv} \\
\& \text{UNCHANGED } \text{appPcr} \\
\& \text{UNCHANGED } \text{semPcr} \\
\& \text{UNCHANGED } \text{bootCtr} \\
\& \text{UNCHANGED } \text{chkptts} \\
\& \text{UNCHANGED } \text{tsvalues}
\end{align*}
\]
Secure execution mode within recovery step 1, when there is an incorrect entry.

\[\text{NextSemRecov}_1\text{ WhenIncorrect} \triangleq \]
\[\land pc = \text{PcRECOV}_1\]
\[\land \neg \text{EnterSemRecovPredicate}\]
\[\land \text{semPcr}' = \text{PcrExtend}(\text{semPcr, PcrxUNHAPPY})\]
\[\land pc' = \text{PcIDLE}\]
\[\land \text{UNCHANGED nv}\]
\[\land \text{UNCHANGED appPcr}\]
\[\land \text{UNCHANGED bootCtr}\]
\[\land \text{UNCHANGED chkptts}\]
\[\land \text{UNCHANGED tsvalues}\]
\[\land \text{UNCHANGED obtains}\]
\[\land \text{UNCHANGED revokes}\]

Secure execution mode within recovery step 2. Record that the \text{nv} app pcr might no longer be current.

\[\text{NextSemRecov}_2 \triangleq \]
\[\text{LET}\]
\[\text{nvcurrent}_1 \triangleq \]
\[\text{IF BugRecovNoClrCur THEN } \text{nv.current} \text{ ELSE FALSE}\]
\[\text{IN}\]
\[\land pc = \text{PcRECOV}_2\]
\[\land \text{nv}' = [\text{nv EXCEPT }.current = \text{nvcurrent}_1]\]
\[\land pc' = \text{PcRECOV}_3\]
\[\land \text{UNCHANGED appPcr}\]
\[\land \text{UNCHANGED semPcr}\]
\[\land \text{UNCHANGED bootCtr}\]
\[\land \text{UNCHANGED chkptts}\]
\[\land \text{UNCHANGED tsvalues}\]
\[\land \text{UNCHANGED obtains}\]
\[\land \text{UNCHANGED revokes}\]

Secure execution mode within recovery step 3. Declare correct recovery happiness and exit secure execution mode.

\[\text{NextSemRecov}_3 \triangleq \]
\[\land pc = \text{PcRECOV}_3\]
\[\land \text{semPcr}' = \text{PcrExtend}(\text{semPcr, PcrxHAPPY})\]
\[\land pc' = \text{PcIDLE}\]
\[\land \text{UNCHANGED nv}\]
Perform a “seal operation” and remember the signed transport session.

In proper execution, provided that the secure execution mode pcr shows that recovery was happy, this action is performed as part of checkpoint during system shutdown. Secure execution mode within checkpoint is then invoked with this transport session as data.

This transport session reads the values of the secure execution mode pcr, the application pcr, and the reboot counter. Then the secure execution mode pcr is extended so that no key bindings will be available until the next happy recovery.

The adversary can record all of the signed transport sessions and try to replay an earlier one to convince secure execution mode within checkpoint to save an old application pcr as “current”. Reading the reboot counter here, and incrementing it in secure execution mode within checkpoint, prevents that.

\[
\text{NextSealTs } \triangleq \\
\text{LET} \\
\quad ts \triangleq \\
\quad \left[ \\
\quad \quad \text{semPcr} \mapsto \text{semPcr,} \quad \text{sem pcr on entry} \\
\quad \quad \text{appPcr} \mapsto \text{appPcr,} \quad \text{app pcr on entry} \\
\quad \quad \text{bootCtr} \mapsto \text{bootCtr} \quad \text{reboot ctr on entry} \\
\quad \right] \\
\quad \text{semPcr}_1 \triangleq \\
\quad \text{IF} \ BugSealNoExt \ \text{THEN} \ semPcr \ \text{ELSE} \\
\quad PcrExtend(semPcr, PcrSEAL) \\
\quad \text{IN} \\
\quad \neg \text{InSem} \quad \text{must not be in secure execution mode} \\
\quad tsvalues' = tsvalues \cup \{ts\} \\
\quad \text{semPcr'} = \text{semPcr}1 \\
\quad \text{UNCHANGED} \ \text{nv} \\
\quad \text{UNCHANGED} \ \text{appPcr} \\
\quad \text{UNCHANGED} \ \text{bootCtr} \\
\quad \text{UNCHANGED} \ \text{pc} \\
\quad \text{UNCHANGED} \ \text{chkptts} \\
\quad \text{UNCHANGED} \ \text{obtains} \\
\quad \text{UNCHANGED} \ \text{revokes}
Enter secure execution mode within checkpoint.

In proper execution, this action is performed during system shutdown following the seal transport session action.

The adversary can perform this action at any idle time, feeding it any known seal transport session value.

\[
NextEnterSemChkpt \triangleq \\
\wedge \neg \text{InSem} \quad \text{must not be in secure execution mode} \\
\wedge \exists ts \in tsvalues : \text{any known ts value} \\
\wedge semPcr' = \text{SemProtect} \\
\wedge pc' = \text{PcCHKPT1} \\
\wedge chkptts' = ts \\
\wedge \text{UNCHANGED nv} \\
\wedge \text{UNCHANGED appPcr} \\
\wedge \text{UNCHANGED bootCtr} \\
\wedge \text{UNCHANGED tsvalues} \\
\wedge \text{UNCHANGED obtains} \\
\wedge \text{UNCHANGED revokes}
\]

Predicate for correct entry to secure execution mode within checkpoint.

\[
EnterSemChkptPredicate \triangleq \\
\wedge \text{chkptts} \in \text{SignedTs} \\
\wedge \text{IF BugChkptNoCheckTsHappy THEN TRUE ELSE} \\
\text{chkptts.semPcr} = \text{SemHappy} \\
\wedge \text{IF BugChkptNoCheckTscsCtr THEN TRUE ELSE} \\
\text{chkptts.bootCt} = \text{bootCt}
\]

Secure execution mode within checkpoint step 1, when there is correct entry.

\[
NextSemChkpt1WhenCorrect \triangleq \\
\wedge pc = \text{PcCHKPT1} \\
\wedge EnterSemChkptPredicate \\
\wedge pc' = \text{PcCHKPT2} \\
\wedge \text{UNCHANGED nv} \\
\wedge \text{UNCHANGED appPcr} \\
\wedge \text{UNCHANGED semPcr} \\
\wedge \text{UNCHANGED bootCt} \\
\wedge \text{UNCHANGED chkptts} \\
\wedge \text{UNCHANGED tsvalues} \\
\wedge \text{UNCHANGED obtains} \\
\wedge \text{UNCHANGED revokes}
\]

Secure execution mode within checkpoint step 1, when there is incorrect entry.
Secure execution mode within checkpoint step 2. Save in \( n^v \) the app ptr recorded at ts entry.

\[
\text{NextSemChkpt2} \triangleq \\
\text{LET } \quad n^v\text{appPcr} \triangleq \\
\text{IF } \text{BugChkptSaveCurApp} \text{ THEN } appPcr \text{ ELSE } \text{chkptts.appPcr} \\
\text{IN} \\
\& pc = Pc\text{CHKPT}2 \\
\& n^v = [n^v \text{ EXCEPT } !.appPcr = n^v\text{appPcr}] \\
\& pc' = Pc\text{CHKPT}3 \\
\& \text{UNCHANGED appPcr} \\
\& \text{UNCHANGED semPcr} \\
\& \text{UNCHANGED bootCtr} \\
\& \text{UNCHANGED chkptts} \\
\& \text{UNCHANGED tsvalues} \\
\& \text{UNCHANGED obtains} \\
\& \text{UNCHANGED revokes}
\]

Secure execution mode within checkpoint step 3. Prevent a ts replay by incrementing the reboot ctr.

\[
\text{NextSemChkpt3} \triangleq \\
\text{LET } \\
\quad bootCtr1 \triangleq \\
\text{IF } \text{BugChkptNoIncCtr} \text{ THEN } bootCtr \text{ ELSE } bootCtr + 1 \\
\text{IN} \\
\& pc = Pc\text{CHKPT}3 \\
\& bootCt' = bootCtr1
\]
\[ pc' = \text{PcCHKPT}4 \]
\[ \land \ \text{UNCHANGED } nv \]
\[ \land \ \text{UNCHANGED } \text{appPcr} \]
\[ \land \ \text{UNCHANGED } \text{semPcr} \]
\[ \land \ \text{UNCHANGED } \text{chkptts} \]
\[ \land \ \text{UNCHANGED } \text{tsvalues} \]
\[ \land \ \text{UNCHANGED } \text{obtains} \]
\[ \land \ \text{UNCHANGED } \text{revokes} \]

Secure execution mode within checkpoint step 4. Declare that the \( nv \ \text{appPcr} \) is current so that after reboot recovery will be able to succeed.

\[ \text{NextSemChkpt}4 \triangleq \]
\[ \text{LET} \]
\[ nvcurrent1 \triangleq \]
\[ \text{It is a bug for secure execution mode within checkpoint to fail to set the } NV \text{ RAM current flag.} \]
\[ \text{Actually, this bug does not result in a safety violation.} \]
\[ \text{IF } \text{BugChkptNoSetCur} \text{ THEN } nv.current \text{ ELSE TRUE} \]
\[ \text{IN} \]
\[ \land pc = \text{PcCHKPT}4 \]
\[ \land nv' = [nv \ \text{EXCEPT } !.current = nvcurrent1] \]
\[ \land pc' = \text{PcCHKPT}5 \]
\[ \land \ \text{UNCHANGED } \text{appPcr} \]
\[ \land \ \text{UNCHANGED } \text{semPcr} \]
\[ \land \ \text{UNCHANGED } \text{bootCtr} \]
\[ \land \ \text{UNCHANGED } \text{chkptts} \]
\[ \land \ \text{UNCHANGED } \text{tsvalues} \]
\[ \land \ \text{UNCHANGED } \text{obtains} \]
\[ \land \ \text{UNCHANGED } \text{revokes} \]

Secure execution mode within checkpoint step 5. Extend sem per with unhappy so protected \( nv \ \text{ram} \) will be inaccessible.

\[ \text{NextSemChkpt}5 \triangleq \]
\[ \land pc = \text{PcCHKPT}5 \]
\[ \land \text{semPcr'} = \text{PcrExtend}(\text{semPcr}, \text{PcrxUNHAPPY}) \]
\[ \land pc' = \text{PcIDLE} \]
\[ \land \ \text{UNCHANGED } nv \]
\[ \land \ \text{UNCHANGED } \text{appPcr} \]
\[ \land \ \text{UNCHANGED } \text{bootCtr} \]
\[ \land \ \text{UNCHANGED } \text{chkptts} \]
\[ \land \ \text{UNCHANGED } \text{tsvalues} \]
\[ \land \ \text{UNCHANGED } \text{obtains} \]
\[ \land \ \text{UNCHANGED } \text{revokes} \]
**SPECIFICATION**

Init $\triangleq$

$\land nv = \text{InitNv}$
$\land \text{appPcr} = \text{AppReboot}$
$\land \text{semPcr} = \text{SemReboot}$
$\land \text{bootCtr} = 0$
$\land \text{pc} = \text{PcIDLE}$
$\land \text{chkptts} = \text{NullTs}$
$\land \text{tsvalues} = \{ \text{NullTs} \}$
$\land \text{obtains} = \{ \}$
$\land \text{revokes} = \{ \}$

Next $\triangleq$

$\lor \text{NextObtainAccess}$
$\lor \text{NextProveRevoke}$
$\lor \text{NextReboot}$
$\lor \text{NextForgetSealTs}$
$\lor \text{NextExtendAppPcr}$
$\lor \text{NextExtendSemPcr}$
$\lor \text{NextIncBootCtr}$
$\lor \text{NextEnterSemRecov}$
$\lor \text{NextSemRecov1WhenCorrect}$
$\lor \text{NextSemRecov1WhenIncorrect}$
$\lor \text{NextSemRecov2}$
$\lor \text{NextSemRecov3}$
$\lor \text{NextSealTs}$
$\lor \text{NextEnterSemChkpt}$
$\lor \text{NextSemChkpt1WhenCorrect}$
$\lor \text{NextSemChkpt1WhenIncorrect}$
$\lor \text{NextSemChkpt2}$
$\lor \text{NextSemChkpt3}$
$\lor \text{NextSemChkpt4}$
$\lor \text{NextSemChkpt5}$

$\text{Spec} \triangleq \text{Init} \land \square[\text{Next}]_{\text{vars}}$

**INvariants**
Type invariant.

\[
\text{InvType} \triangleq \\
\land \nv :: \nv \in Nv \\
\land \text{appPcr} :: \text{appPcr} \in \text{Pcr} \\
\land \text{semPcr} :: \text{semPcr} \in \text{Pcr} \\
\land \text{bootCtr} :: \text{bootCtr} \in \text{Nat} \\
\land \text{pc} :: \text{pc} \in \text{Pc} \\
\land \text{chkptts} :: \land \text{chkptts} \in \text{Ts} \\
\land \text{pc} \in \text{PcChkpt}\{\text{PcCHKPT}1\} \Rightarrow \text{chkptts} \in \text{SignedTs} \\
\land \text{tsvalues} :: \text{tsvalues} \in \text{SUBSET} \text{Ts} \\
\land \text{obtains} :: \text{obtains} \in \text{SUBSET} \text{Pcr} \\
\land \text{revokes} :: \text{revokes} \in \text{SUBSET} \text{Pcr}
\]

\[
Nv \text{ protection invariant.}
\]

Being in secure execution mode is equivalent to saying that the secure execution mode \text{pcr} permits access to protected \text{Nv} ram.

\[
\text{InvNvProtection} \triangleq \\
\text{InSem} \equiv (\text{semPcr} = \text{SemProtect})
\]

Verifiable revocation invariant. There had better not be any decisions to obtain access for which a proof of revocation was also constructed.

\[
\text{InvVerifiableRevocation} \triangleq \\
\forall \text{obtains} : \text{last extension was OBTAIN} \\
\forall \text{revokes} : \text{last extension was REVOKE} \\
\text{PcrPrior}(o) \neq \text{PcrPrior}(r) \quad \text{cannot have both extended from same place}
\]

Access undeniability.

This invariant is modeled as performing an audit on the present state and seeing that all key bindings that have been used to obtain access appear in the audit report. A key binding \(o\) appears in the audit report iff \(\text{PcrLeq}(o, \text{appPcr})\), which means that there exists a sequence of zero or more extensions from \(o\) that reach \(\text{appPcr}\).

However, it might be impossible to generate a valid audit report in the present node state. That is okay.

\[
\text{InvAccessUndeniability} \triangleq \\
\text{It is a bug to fail to require the audit to quote } \text{SemHappy}.
\]

\[
\land \quad \text{IF } \text{BugAuditNoCheckHappy} \text{ THEN TRUE ELSE} \\
\text{semPcr} = \text{SemHappy} \\
\Rightarrow \\
\forall \text{obtains} : \text{PcrLeq}(o, \text{appPcr})
\]
E Proof of Optimized Pasture

---

**MODULE** *OptPastureNodeProof*

**EXTENDS** *OptPastureNode*, *TLAPS*

---

**STATE FUNCTIONS**

We talk about the “log” being in various places. Actually, what is in those places is a cryptographic summary of the log, which is of type *Pcr*. However, under the anticollision assumption of *PcrExtend*, the cryptographic summary is effectively in one-to-one correspondence with the actual log. So we talk as if the cryptographic summary were the log, rather than merely a reference to the log.

Check that *ts* is a valid seal attestation in the current node state. To be valid it must be a signed attestation and it must record *SemHappy* and the current boot counter.

\[
\text{CheckTsIsCurrent}(ts) \triangleq \\
\land ts \in \text{SignedTs} \\
\land ts.\text{semPcr} = \text{SemHappy} \\
\land ts.\text{bootCtr} = \text{bootCtr}
\]

All valid seal attestations in the current node state. Seal attestations can be found among the known values (in *tsvalues*) or in the temporary state variable *chkptts* used during the checkpoint sem routine.

\[
\text{AllCurrentTs} \triangleq \\
\{ ts \in \text{tsvalues} \cup \{ \text{chkptts} \} : \text{CheckTsIsCurrent}(ts) \}
\]

If there are any valid seal attestations in the current node state, choose one and get its log.

\[
\text{CurrentTsLog} \triangleq \\
\text{LET } ts \triangleq \text{CHOOSE } ts \in \text{AllCurrentTs} : \text{TRUE} \\
\text{IN } ts.\text{appPcr}
\]

The log is present in the *nv* ram. This is true iff the *nv* ram says it is current.

\[
\text{LogInNv} \triangleq \\
\land \text{nv.current}
\]

The log is present in the application pcr. This is true iff the sem pcr contains *SemHappy*.

\[
\text{LogInApp} \triangleq \\
\land \text{semPcr} = \text{SemHappy}
\]

The log is present in some known seal attestation. This is true iff there exists a valid seal attestation in the current node state.

\[
\text{LogInTs} \triangleq \\
\]
AllCurrentTs \neq \{\}

Assuming the log exists, determine if \( \text{Pcr } p \) is on it.

The log has a domicile in the \( \text{nv} \) ram, when the \( \text{nv} \) ram is marked as current. The log has a domicile in the application \( \text{pcr} \) when the \( \text{sem pcr} \) contains \( \text{SemHappy} \). The log has a domicile in a seal ts attestation when that attestation quotes \( \text{SemHappy} \) and the current bootCtr.

During secure execution mode, the log can also temporarily live in certain places, as it is moved from one domicile to another.

\[ \text{IsOnLog}(p) \triangleq \]

Where to find the log at most times.

\begin{align*}
\lor \text{LogInNv} & \Rightarrow \text{PcrLeq}(p, \text{nv.appPcr}) \\
\lor \text{LogInApp} & \Rightarrow \text{PcrLeq}(p, \text{appPcr}) \\
\lor \text{LogInTs} & \Rightarrow \text{PcrLeq}(p, \text{CurrentTsLog})
\end{align*}

Special places to find the log during secure execution mode.

\begin{align*}
\lor p = \text{PcrRECOV1} & \Rightarrow \text{TRUE} \\
\lor p = \text{PcrRECOV2} & \Rightarrow \text{PcrLeq}(p, \text{appPcr}) \\
\lor p = \text{PcrRECOV3} & \Rightarrow \text{PcrLeq}(p, \text{appPcr}) \\
\lor p = \text{PcrCHKPT1} & \Rightarrow \text{TRUE} \\
\lor p = \text{PcrCHKPT2} & \Rightarrow \text{PcrLeq}(p, \text{chkptts.appPcr}) \\
\lor p = \text{PcrCHKPT3} & \Rightarrow \text{PcrLeq}(p, \text{nv.appPcr}) \\
\lor p = \text{PcrCHKPT4} & \Rightarrow \text{PcrLeq}(p, \text{nv.appPcr}) \\
\lor p = \text{PcrCHKPT5} & \Rightarrow \text{PcrLeq}(p, \text{nv.appPcr})
\end{align*}

---

**ADDITIONAL INVARIANTS**

When the node is in secure execution mode, the secure execution mode \( \text{pcr} \) contains \( \text{SemProtect} \).

\[ \text{InvInSemProtect} \triangleq \]

\begin{align*}
\lor \text{InvType} \\
\lor \text{goal::} \\
\text{InSem} & \Rightarrow \text{semPcr} = \text{SemProtect}
\end{align*}

When the node is not in secure execution mode, the secure execution mode \( \text{pcr} \) contains a value from which \( \text{SemProtect} \) cannot be reached.

\[ \text{InvUnreachableSemProtect} \triangleq \]

\begin{align*}
\lor \text{InvType} \\
\lor \text{InvInSemProtect} \\
\lor \text{goal::} \\
\neg \text{InSem} & \Rightarrow \neg \text{PcrLeq}(\text{semPcr}, \text{SemProtect})
\end{align*}
All known signed seal attestations quote a bootCtr that does not exceed the current bootCtr.

\[
\text{InvSignedTsLeqBoot} \triangleq \\
\land \text{InvType} \\
\land \text{goal}:: \\
\forall ts \in tvalues \cup \{\text{chkptTs}\} : \\
ts \in \text{SignedTs} \Rightarrow ts.\text{bootCtr} \leq \text{bootCtr}
\]

When the node is not in secure execution mode, the secure execution mode pcr contains either (1) SemHappy or (2) a value from which SemHappy cannot be reached.

\[
\text{InvUnforgeableSemHappy} \triangleq \\
\land \text{InvType} \\
\land \text{InvInSemProtect} \\
\land \text{goal}:: \\
\neg \text{InSem} \Rightarrow \\
\lor \text{semPcr} = \text{SemHappy} \\
\lor \neg \text{PcrLeq(semPcr, SemHappy)}
\]

Every entry in obtains and revokes has a last extension of OBTAIN and REVOKE, respectively.

\[
\text{InvProperLastExtension} \triangleq \\
\land \text{InvType} \\
\land \text{goal}:: \\
\land \forall o \in \text{obtains} : \text{PcrHasExtension}(o) \land \text{PcrLastExtension}(o) = \text{PcrxOBTAIN} \\
\land \forall r \in \text{revokes} : \text{PcrHasExtension}(r) \land \text{PcrLastExtension}(r) = \text{PcrxREVOKE}
\]

There is at most one log.

\[
\text{InvOneLog} \triangleq \\
\land \text{InvType} \\
\land \text{InvSignedTsLeqBoot} \\
\land \text{InvInSemProtect} \\
\land \text{InvUnforgeableSemHappy} \\
\land \text{InvProperLastExtension} \\
\land \text{goal}::
\]

(One Log to rule them all, One Log to find them, One Log to bring them all and in the darkness bind them. (with apologies to J. R. R. Tolkien))

\[
\text{InvOneLog} \triangleq \\
\land \text{InvType} \\
\land \text{InvSignedTsLeqBoot} \\
\land \text{InvInSemProtect} \\
\land \text{InvUnforgeableSemHappy} \\
\land \text{InvProperLastExtension} \\
\land \text{goal}::
\]
The log can only have at most one domicile at a time.
\[
\begin{align*}
    \text{LogInNv} & \Rightarrow \neg \text{LogInApp} \land \neg \text{LogInTs} \\
    \text{LogInApp} & \Rightarrow \neg \text{LogInNv} \land \neg \text{LogInTs} \\
    \text{LogInTs} & \Rightarrow \neg \text{LogInNv} \land \neg \text{LogInApp}
\end{align*}
\]

Extra requirements during secure execution mode.
\[
\begin{align*}
    \text{pc} = \text{PcRECOV1} & \Rightarrow \text{TRUE} \\
    \text{pc} = \text{PcRECOV2} & \Rightarrow \text{LogInNv} \\
    \text{pc} = \text{PcRECOV3} & \Rightarrow \neg \text{LogInNv} \land \neg \text{LogInApp} \land \neg \text{LogInTs} \\
    \text{pc} = \text{PcCHKPT1} & \Rightarrow \text{TRUE} \\
    \text{pc} = \text{PcCHKPT2} & \Rightarrow \text{LogInTs} \land \text{CheckTsIsCurrent(chkptts)} \\
    \text{pc} = \text{PcCHKPT3} & \Rightarrow \text{LogInTs} \land \text{CheckTsIsCurrent(chkptts)} \\
    \text{pc} = \text{PcCHKPT4} & \Rightarrow \neg \text{LogInNv} \land \neg \text{LogInApp} \land \neg \text{LogInTs} \\
    \text{pc} = \text{PcCHKPT5} & \Rightarrow \text{LogInNv}
\end{align*}
\]

All seal attestations containing the log must have the same log.
\[
\begin{align*}
    \forall ts1, ts2 \in \text{AllCurrentTs} : ts1.appPcr & = ts2.appPcr
\end{align*}
\]

Every entry in obtains (a decision to obtain access) is recorded on the log (assuming there is one).
\[
\begin{align*}
    \text{obtains}\text{::} \forall o \in \text{obtains} : \text{IsOnLog}(o)
\end{align*}
\]

Every entry in revokes (a decision to prove revocation) is recorded on the log (assuming there is one).
\[
\begin{align*}
    \text{revokes}\text{::} \forall r \in \text{revokes} : \text{IsOnLog}(r)
\end{align*}
\]

We have verifiable revocation.
\[
\begin{align*}
    \text{InvVerifiableRevocation}
\end{align*}
\]

NECESSARY FACTS ABOUT NATURALS

The SMT prover can prove these easily enough in isolation, but if you ask it to prove them in the middle of other proofs where records and other complicated things are flying around, it usually aborts with a type inference failure.

\[
\leq \text{is a total order}
\]

THEOREM ThmNatLeqIsTotal \(\triangleq \forall i, j \in \text{Nat} : i \leq j \lor j \leq i\) \ BY SMT

THEOREM ThmNatLeqIsReflexive \(\triangleq \forall i \in \text{Nat} : i \leq i\) \ BY SMT

THEOREM ThmNatLeqIsAntisymmetric \(\triangleq \forall i, j \in \text{Nat} : i \leq j \land j \leq i \Rightarrow i = j\) \ BY SMT

THEOREM ThmNatLeqIsTransitive \(\triangleq \forall i, j, k \in \text{Nat} : i \leq j \land j \leq k \Rightarrow i \leq k\) \ BY SMT

\[
\leq \text{minimum is 0}
\]

THEOREM ThmNatLeqMinIsZero \(\triangleq \forall i \in \text{Nat} : 0 \leq i\) \ BY SMT

\[
\leq \text{is the opposite of >}
\]

THEOREM ThmNatLeqXorGt \(\triangleq \forall i, j \in \text{Nat} : i \leq j \equiv \neg (i > j)\) \ BY SMT

THEOREM ThmNatMore \(\triangleq \forall i, j \in \text{Nat} : i \leq i + j\) \ BY SMT
THEOREM ThmNatLess $\triangleq \forall i, j \in \text{Nat} : i - j \leq i$ BY SMT
THEOREM ThmNatInc $\triangleq \forall i \in \text{Nat} : i + 1 > i$ BY SMT
THEOREM ThmNatDotDot $\triangleq \forall i, j, k \in \text{Nat} : i \leq j \land j \leq k \equiv j \in i \ldots k$ BY SMT
THEOREM ThmNatDecZero $\triangleq \forall n \in \text{Nat} : n > 0 \Rightarrow n - 1 \in \text{Nat}$ BY SMT
THEOREM ThmNatAddEq $\triangleq \forall i, j, k \in \text{Nat} : i + k = j + k \Rightarrow i = j$ BY SMT
THEOREM ThmNatLeqLt $\triangleq \forall i, j, k \in \text{Nat} : i \leq j \land j < k \Rightarrow i < k$ BY SMT

NECESSARY FACTS ABOUT SEQUENCES
I have not been able to figure out how to convince the prover to prove most of these.

Definition of a sequence.

THEOREM ThmSeqDef $\triangleq$
ASSUME
  NEW CONSTANT $S$, 
  NEW CONSTANT $q \in \text{Seq}(S)$
PROVE
  $q = [i \in 1 \ldots \text{Len}(q) \mapsto q[i]]$
PROOF
  OMITTED

The empty sequence is a sequence of $S$, for any $S$.

THEOREM ThmSeqEmptyIsSeq $\triangleq$
ASSUME
  NEW CONSTANT $S$
PROVE
  $\langle \rangle \in \text{Seq}(S)$
PROOF
  OMITTED

For any sequence $q$ of $S$, $\text{Len}(q) \in \text{Nat}$.

THEOREM ThmSeqLenIsNat $\triangleq$
ASSUME
  NEW CONSTANT $S$, 
  NEW $q \in \text{Seq}(S)$
PROVE
\[ Len(q) \in \mathbb{Nat} \]
PROOF
OMITTED

For any non-empty sequence of \( S \), its tail is a sequence of \( S \).

**THEOREM ThmSeqTailIsSeq**

ASSUME
NEW CONSTANT \( S \),
NEW CONSTANT \( q \in \text{Seq}(S) \),
\( q \neq \langle \rangle \)
PROVE
\( \text{Tail}(q) \in \text{Seq}(S) \)
PROOF
OMITTED

For any sequence of \( S \), appending \( x \in S \) yields a sequence of \( S \).

**THEOREM ThmSeqAppendIsSeq**

ASSUME
NEW CONSTANT \( S \),
NEW CONSTANT \( q \in \text{Seq}(S) \),
NEW CONSTANT \( x \in S \)
PROVE
\( \text{Append}(q, x) \in \text{Seq}(S) \)
PROOF
OMITTED

The result of \( \text{Append}(q, x) \) is one longer than \( q \).

**THEOREM ThmSeqAppendLen1**

ASSUME
NEW CONSTANT \( S \),
NEW CONSTANT \( q \in \text{Seq}(S) \),
NEW CONSTANT \( x \in S \)
PROVE
\[ \text{Len}(\text{Append}(q, x)) = \text{Len}(q) + 1 \]
PROOF
OMITTED

The result of \( \text{Append}(q, x) \) starts with \( q \).

**THEOREM ThmSeqAppendSubSeq**

ASSUME
NEW CONSTANT $S$,  
NEW CONSTANT $q \in \text{Seq}(S)$,  
NEW CONSTANT $x \in S$

PROVE  
$\text{SubSeq}(\text{Append}(q, x), 1, \text{Len}(q)) = q$

PROOF  
OMITTED

Appending the last entry onto all but the last of a sequence yields the original sequence.

THEOREM ThmSeqAppendPriorLast $\triangleq$

ASSUME  
NEW CONSTANT $S$,  
NEW CONSTANT $q \in \text{Seq}(S)$,  
$\text{Len}(q) > 0$

PROVE  
$\text{Append}(\text{SubSeq}(q, 1, \text{Len}(q) - 1), q[\text{Len}(q)]) = q$

PROOF  
OMITTED

The entire initial $\text{SubSeq}$ of $q$ is $q$.

THEOREM ThmSeqEntireInitialSubSeq $\triangleq$

ASSUME  
NEW CONSTANT $S$,  
NEW CONSTANT $q \in \text{Seq}(S)$

PROVE  
$q = \text{SubSeq}(q, 1, \text{Len}(q))$

PROOF  
OMITTED

Initial $\text{SubSeq} \in \text{sequence}$.

THEOREM ThmSeqInitialSubSeqIsSeq $\triangleq$

ASSUME  
NEW CONSTANT $S$,  
NEW CONSTANT $q \in \text{Seq}(S)$,  
NEW CONSTANT $n \in \text{Nat}$,  
$n \leq \text{Len}(q)$

PROVE  
$\text{SubSeq}(q, 1, n) \in \text{Seq}(S)$

PROOF  
OMITTED
Initial $SubSeq$ is antisymmetric.

**THEOREM** $ThmSeqInitialSubSeqIsAntisymmetric$ \(\triangleq\)

**ASSUME**
- NEW CONSTANT $S$,
- NEW CONSTANT $q \in Seq(S)$,
- NEW CONSTANT $r \in Seq(S)$,
- $Len(q) \leq Len(r)$,
- $Len(r) \leq Len(q)$,
- $q = SubSeq(r, 1, Len(q))$,
- $r = SubSeq(q, 1, Len(r))$

**PROVE**
- $q = r$

**PROOF**

\((1)\) $Len(q) = Len(r)$
\((2)\) USE $ThmSeqLenIsNat$
\((2)\) QED BY $ThmNatLeqIsAntisymmetric$
\((1)\) QED BY $ThmSeqEntireInitialSubSeq$

Initial $SubSeq$ is transitive.

**THEOREM** $ThmSeqInitialSubSeqIsTransitive$ \(\triangleq\)

**ASSUME**
- NEW CONSTANT $S$,
- NEW CONSTANT $q \in Seq(S)$,
- NEW CONSTANT $r \in Seq(S)$,
- NEW CONSTANT $s \in Seq(S)$,
- $Len(q) \leq Len(r)$,
- $Len(r) \leq Len(s)$,
- $q = SubSeq(r, 1, Len(q))$,
- $r = SubSeq(s, 1, Len(r))$

**PROVE**
- $q = SubSeq(s, 1, Len(q))$

**PROOF**

OMITTED

Sequence append incompatible.

**THEOREM** $ThmSeqAppendIncompatible$ \(\triangleq\)

**ASSUME**
- NEW CONSTANT $S$,
- NEW CONSTANT $q \in Seq(S)$,
- NEW CONSTANT $s1 \in S$,
- NEW CONSTANT $s2 \in S$,
- $s1 \neq s2$
PROVE
\( \text{Append}(q, s_1) \neq \text{Append}(q, s_2) \)

PROOF
OMITTED

Sequence append anti-collision.

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

**ASSUME**
- NEW CONSTANT \( S \),
- NEW CONSTANT \( q \),
- NEW CONSTANT \( q_1 \in \text{Seq}(S) \),
- NEW CONSTANT \( q_2 \in \text{Seq}(S) \),
- NEW CONSTANT \( s_1 \in S \),
- NEW CONSTANT \( s_2 \in S \),
- \( q = \text{Append}(q_1, s_1) \),
- \( q = \text{Append}(q_2, s_2) \)

**PROVE**
\( q_1 = q_2 \land s_1 = s_2 \)

**PROOF**
OMITTED

---

**PCR OPERATOR THEOREMS**

**PcrInit \( \in \text{Pcr} \)**

**THEOREM** \( \text{ThmPcrInitIsPcr} \) \( \triangleq \)
\( \forall i \in \text{Pcri} : \text{PcrInit}(i) \in \text{Pcr} \)

**PROOF**
(1) TAKE \( i \in \text{Pcri} \)
(1) USE DEF \( \text{Pcr, PcrInit} \)
(1) DEFINE \( p \triangleq \text{PcrInit}(i) \)
(1)1. \( p\).\text{init} \( \in \text{PeriOBVIOUS} \)
(1)2. \( p\).\text{extq} \( \in \text{Seq(Pcr)} \) BY \( \text{ThmSeqEmptyIsSeq} \)
(1) QED BY (1)1, (1)2

**PcrExtend \( \in \text{Pcr} \)**
THEOREM ThmPcrExtendIsPcr \triangleq \\
\forall p \in Pcr, x \in Pcr : PcrExtend(p, x) \in Pcr \\
PROOF \\
(1) TAKE p \in Pcr, x \in Pcr \\
(1) USE DEF Pcr, PcrExtend \\
(1) DEFINE px \triangleq PcrExtend(p, x) \\
(1)1. px.init \in \text{PerIOBVIOUS} \\
(1)2. px.extq \in \text{Seq}(Pcrx) BY ThmSeqAppendIsSeq \\
(1) QED BY (1)1, (1)2 \\

\text{PcrLen} \in \text{Nat} \\
THEOREM ThmPcrLenIsNat \triangleq \\
\forall p \in Pcr : PcrLen(p) \in \text{Nat} \\
PROOF \\
(1) TAKE p \in Pcr \\
(1) USE DEF PcrLen \\
(1) USE DEF Pcr \\
(1) QED BY ThmSeqLenIsNat \\

p \leq PcrExtend(p, x) \\
THEOREM ThmPcrExtendLeq \triangleq \\
\forall p \in Pcr, x \in Pcrx : \text{PcrLeq}(p, PcrExtend(p, x)) \\
PROOF \\
(1) TAKE p \in Pcr, x \in Pcrx \\
(1) DEFINE px \triangleq PcrExtend(p, x) \\
(1) USE DEF Pcr \\
(1) USE DEF PcrExtend \\
(1) USE DEF PcrLeq \\
(1)1. p.init = px.init \text{OBVIOUS} \\
(1)2. \text{Len}(p.extq) \leq \text{Len}(px.extq) \\
\langle 2 \rangle \text{Len}(p.extq) + 1 = \text{Len}(px.extq) \text{BY ThmSeqAppendLen1} \\
\langle 2 \rangle \USE \text{ThmSeqLenIsNat} \\
\langle 2 \rangle QED BY ThmNatMore \\
\langle 1 \rangle3. p.extq = \text{SubSeq}(px.extq, 1, \text{Len}(p.extq)) \text{BY ThmSeqAppendSubSeq} \\
\langle 1 \rangle QED BY (1)1, (1)2, (1)3 \\

p \neq PcrExtend(p, x) \\
THEOREM ThmPcrExtendNeq \triangleq \\
\forall p \in Pcr, x \in Pcrx : p \neq PcrExtend(p, x) \\
PROOF \\
(1) TAKE p \in Pcr, x \in Pcrx \\
(1) DEFINE px \triangleq PcrExtend(p, x) \\
(1) USE DEF Pcr
130  E  PROOF OF OPTIMIZED PASTURE

\( Pcr \) equality. This would seem to be trivial but the prover cannot seem to figure it out by itself.

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

\[ \forall p, q \in Pcr : \\
\wedge p.\init = q.\init \\
\wedge p.\extq = q.\extq \\
\Rightarrow p = q \]

**PROOF**

(1) \text{TAK}\text{E}  \( p, q \in Pcr \)

(1) \text{HAY}  \( p.\init = q.\init \wedge p.\extq = q.\extq \)

(1) \text{USE DEF } Pcr

The following fact seems to be necessary to help the prover.

(1) \( p = [q \text{ EXCEPT } \!.\init = p.\init, \!.\extq = p.\extq] \text{OBVIOUS} \)

(1) \text{QED OBVIOUS}

**Anti-collision property.**

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

\[ \forall p1, p2 \in Pcr, x1, x2 \in Pcrx : \\
Pcr\text{Extend}(p1, x1) = Pcr\text{Extend}(p2, x2) \Rightarrow p1 = p2 \wedge x1 = x2 \]

**PROOF**

(1) \text{TAK}\text{E}  \( p1, p2 \in Pcr, x1, x2 \in Pcrx \)

(1) \text{DEFINE } px1 \triangleq Pcr\text{Extend}(p1, x1)

(1) \text{DEFINE } px2 \triangleq Pcr\text{Extend}(p2, x2)

(1) \text{HAVE } px1 = px2

(1) \text{USE DEF } Pcr

(1) \text{USE DEF } Pcr\text{Extend}

(1) \text{QED}

(2) \( p1.\init = p2.\init \text{OBVIOUS} \)
Create definitions for the $\text{ext}_q$ fields and then hide them to prevent overwhelming the prover.

\[
\text{DEFINE } p_1 q \triangleq p_1 . \text{ext}_q \\
\text{DEFINE } p_2 q \triangleq p_2 . \text{ext}_q \\
\text{HIDE DEF } p_1 q \\
\text{HIDE DEF } p_2 q \\
\text{Append}(p_1 q, x_1) = \text{Append}(p_2 q, x_2) \text{ BY DEF } p_1 q, p_2 q \\
p_1 q \in \text{Seq}(\text{Pcr}_x) \text{ BY DEF } p_1 q \\
p_2 q \in \text{Seq}(\text{Pcr}_x) \text{ BY DEF } p_2 q \\
p_1 q = p_2 q \land x_1 = x_2 \text{ BY ThmSeqAppendAnticollision} \\
\text{QED BY DEF } p_1 q, p_2 q \\
\text{QED BY ThmPcrEqual}
\]

If two extensions of the same pcr are both $\leq$ a target pcr, then the extensions must be the same.

**THEOREM** ThmPcrExtendLeqAnticollision $\triangleq$

\[\forall p, t \in \text{Pcr}, x_1, x_2 \in \text{Pcr}_x : \text{PcrLeq}(\text{PcrExtend}(p, x_1), t) \land \text{PcrLeq}(\text{PcrExtend}(p, x_2), t) \Rightarrow x_1 = x_2\]

**PROOF**

\[\text{TAKEN } p, t \in \text{Pcr}, x_1, x_2 \in \text{Pcr}_x\]

\[\text{HAVE } \text{PcrLeq}(\text{PcrExtend}(p, x_1), t) \land \text{PcrLeq}(\text{PcrExtend}(p, x_2), t)\]

\[\text{USE DEF Pcr}\]

\[\text{USE DEF PcrExtend}\]

\[\text{USE DEF PcrLeq}\]

\[\text{DEFINE } q p \triangleq p . \text{ext}_q\]

\[\text{DEFINE } q t \triangleq t . \text{ext}_q\]

\[\text{DEFINE } q p x_1 \triangleq \text{Append}(q p, x_1)\]

\[\text{DEFINE } q p x_2 \triangleq \text{Append}(q p, x_2)\]

\[q p \in \text{Seq}(\text{Pcr}_x) \text{ OBVIOUS}\]

\[q t \in \text{Seq}(\text{Pcr}_x) \text{ OBVIOUS}\]

\[\text{Len}(q p x_1) \leq \text{Len}(q t) \text{ OBVIOUS}\]

\[\text{Len}(q p x_2) \leq \text{Len}(q t) \text{ OBVIOUS}\]

\[\text{SubSeq}(q t, 1, \text{Len}(q p x_1)) = q p x_1 \text{ OBVIOUS}\]

\[\text{SubSeq}(q t, 1, \text{Len}(q p x_2)) = q p x_2 \text{ OBVIOUS}\]

\[\text{HIDE DEF } q p\]

\[\text{HIDE DEF } q t\]

\[\text{Len}(q p x_1) = \text{Len}(q p) + 1 \text{ BY ThmSeqAppendLen}\]

\[\text{Len}(q p x_2) = \text{Len}(q p) + 1 \text{ BY ThmSeqAppendLen}\]

\[\text{Len}(q p x_1) = \text{Len}(q p x_2) \text{ OBVIOUS}\]

\[q p x_1 = q p x_2 \text{ OBVIOUS}\]

\[\text{QED}\]

The prover really needs help to focus its attention.

\[\text{1. } x_1 \in \text{Pcr}_x \text{ OBVIOUS}\]

\[\text{2. } x_2 \in \text{Pcr}_x \text{ OBVIOUS}\]

\[\text{3. } q p \in \text{Seq}(\text{Pcr}_x) \text{ OBVIOUS}\]
Append(q, x1) = Append(q, x2) OBVIOUS
(2) QED BY ONLY (2)1, (2)2, (2)3, (2)4, ThmSeqAppendAnticollision

PcrExtend increases the length by 1.

THEOREM ThmPcrExtendLen
\[ \forall p \in Pcr, x \in Pcr : PcrLen(PcrExtend(p, x)) = PcrLen(p) + 1 \]

PROOF
(1) TAKE p \in Pcr, x \in Pcr
(1) DEFINE px = PcrExtend(p, x)
(1) px \in Pcr BY ThmPcrExtendIsPcr
(1) USE DEF PcrLen
(1) USE DEF PcrExtend
(1) USE DEF Pcr
(1) QED BY ThmSeqAppendLen

PcrLeq implies \( \leq \) on respective PcrLen.

THEOREM ThmPcrLeqLeq
\[ \forall p, q \in Pcr : PcrLeq(p, q) \Rightarrow PcrLen(p) \leq PcrLen(q) \]

PROOF
(1) TAKE p, q \in Pcr
(1) HAVE PcrLeq(p, q)
(1) USE DEF Pcr
(1) USE DEF Pcr
(1) Len(p.extq) \leq Len(q.extq) BY DEF PcrLeq
(1) PcrLen(p) = Len(p.extq) BY DEF PcrLen
(1) PcrLen(q) = Len(q.extq) BY DEF PcrLen
(1) QED OBVIOUS

PcrLeq is a partial order.

THEOREM ThmPcrLeqIsReflexive
\[ \forall p \in Pcr : PcrLeq(p, p) \]

PROOF
(1) TAKE p \in Pcr
(1) USE DEF PcrLeq
(1) USE DEF Pcr
(1)1. p.init = p.init OBVIOUS
(1)2. Len(p.extq) \leq Len(p.extq)
(2) USE ThmSeqLenIsNat
(2) USE ThmNatLeqIsReflexive
(2) QED OBVIOUS
THEOREM ThmPcrLeqIsAntisymmetric

\[ \forall p, q \in Pcr : PcrLeq(p, q) \land PcrLeq(q, p) \Rightarrow p = q \]

PROOF

\begin{enumerate}
\item TAKE p, q \in Pcr
\item HAVE PcrLeq(p, q) \land PcrLeq(q, p)
\item p.init = q.init
\item USE DEF PcrLeq
\item QED BY \(1)2\)
\item p.extq = q.extq
\item USE DEF PcrLeq
\item USE DEF Pcr
\item USE ThmSeqInitialSubSeqIsAntisymmetric
\item QED BY \(1)2\)
\item QED
\item USE ThmPcrEqual
\item QED BY \(1)3, \(1)4\)
\end{enumerate}

THEOREM ThmPcrLeqIsTransitive

\[ \forall p, q, r \in Pcr : PcrLeq(p, q) \land PcrLeq(q, r) \Rightarrow PcrLeq(p, r) \]

PROOF

\begin{enumerate}
\item TAKE p, q, r \in Pcr
\item HAVE PcrLeq(p, q) \land PcrLeq(q, r)
\item USE DEF PcrLeq
\item p.init = r.init OBVIOUS
\item Len(p.extq) \leq Len(r.extq)
\item USE DEF Pcr
\item USE ThmSeqLenIsNat
\item QED BY ThmNatLeqIsTransitive
\item p.extq = SubSeq(r.extq, 1, Len(p.extq))
\item USE DEF Pcr
\item QED BY ThmSeqInitialSubSeqIsTransitive
\item QED BY \(1)1, \(1)2, \(1)3\)
\end{enumerate}

An extension of a Pcr p cannot reach p.

THEOREM ThmPcrExtendSelfUnreachable

\[ \forall p \in Pcr, x \in Pcrx : \neg PcrLeq(PcrExtend(p, x), p) \]

PROOF

\begin{enumerate}
\item TAKE p \in Pcr, x \in Pcrx
\item DEFINE px \(=\) PcrExtend(p, x)
\end{enumerate}
(1) Define \( \text{isleq} \triangleq \text{PcrLeq}(px, p) \)

\( \text{isleq} \) by contradiction.

(1) \( \text{isleq} \) \( \triangleq \text{PcrLeq}(px, p) \)

(1) \( px \in \text{Pcr} \) by ThmPcrExtendIsPcr

\[ \text{PcrLeq}(px, q) \]

\( \text{isleq} \) \( \triangleq \text{PcrLeq}(px, p) \)

(1) \( px \in \text{Pcr} \) by ThmPcrExtendIsPcr

Proof by contradiction.

(1) \( \text{isleq} \) \( \triangleq \text{PcrLeq}(px, p) \)

(1) \( px \in \text{Pcr} \) by ThmPcrExtendIsPcr

(1) \( \text{isleq} \) \( \triangleq \text{PcrLeq}(px, p) \)

\[ \text{PcrLeq}(px, q) \]

If an extension of a \( \text{Pcr} \) can reach a target, the \( \text{Pcr} \) itself can reach the target.

**THEOREM** ThmPcrReachableIfExtend \( \triangleq \)

\[ \forall p, q \in \text{Pcr}, x \in \text{Pcr} \Rightarrow \text{PcrLeq}(\text{PcrExtend}(p, x), q) \Rightarrow \text{PcrLeq}(p, q) \]

**PROOF**

(1) \( \text{take} p, q \in \text{Pcr}, x \in \text{Pcr} \)

(1) \( \text{define} px \triangleq \text{PcrExtend}(p, x) \)

(1) \( \text{have} \) \( \text{PcrLeq}(px, q) \)

(1) \( px \in \text{Pcr} \) by ThmPcrExtendIsPcr

(1) \( \text{PcrLeq}(p, px) \) by ThmPcrExtendLeq

(1) \( \text{QED by ThmPcrLeqIsTransitive} \)

If a target \( \text{Pcr} \) is not reachable from a source \( \text{Pcr} \), then it is not reachable from an extension of the source \( \text{Pcr} \).

**THEOREM** ThmPcrExtendSourceUnreachable \( \triangleq \)

\[ \forall p, q \in \text{Pcr}, x \in \text{Pcr} \Rightarrow \neg \text{PcrLeq}(p, q) \Rightarrow \neg \text{PcrLeq}(\text{PcrExtend}(p, x), q) \]

**PROOF**

(1) \( \text{QED by ThmPcrReachableIfExtend} \)

If \( p \) equals \( q \) or cannot reach \( q \), then an extension of \( p \) cannot reach \( q \).
**THEOREM** ThmPcrExtendFromEqOrNotleq ∆

\( \forall p, q \in Pcr, x \in Pcr : \\
p = q \lor \neg PcrLeq(p, q) \Rightarrow \neg PcrLeq(PcrExtend(p, x), q) \)

**PROOF**

1. TAKE \( p, q \in Pcr, x \in Pcr \)
2. HAVE \( p = q \lor \neg PcrLeq(p, q) \)
   
1. CASE \( p = q \) BY ThmPcrExtendSelfUnreachable
2. CASE \( \neg PcrLeq(p, q) \) BY ThmPcrReachableIfExtend

(1) QED OBVIOUS

Different extensions of a Pcr are incompatible.

**THEOREM** ThmPcrExtendIncompatible ∆

\( \forall p \in Pcr, x_1, x_2 \in Pcr : \\
x_1 \neq x_2 \Rightarrow \neg PcrLeq(PcrExtend(p, x_1), PcrExtend(p, x_2)) \)

**PROOF**

1. TAKE \( p \in Pcr, x_1, x_2 \in Pcr \)
2. HAVE \( x_1 \neq x_2 \)
3. DEFINE \( p_1 \triangleq PcrExtend(p, x_1) \)
4. DEFINE \( p_2 \triangleq PcrExtend(p, x_2) \)
5. 1. CASE \( \neg PcrLeq(p_1, p_2) \) BY (1)1
6. 2. CASE \( PcrLeq(p_1, p_2) \) BY (1)2
   
2. USE (1)2
3. USE DEF PcrLeq
4. USE DEF PcrExtend
5. USE DEF Pcr
6. \( p_1.extq \in Seq(Pcr) \) BY ThmSeqAppendIsSeq
7. \( p_2.extq \in Seq(Pcr) \) BY ThmSeqAppendIsSeq
8. \( Len(p_1.extq) = Len(p_2.extq) \)
9. 3. USE (1)1
10. 3. USE (1)2
11. QED OBVIOUS
12. 2. USE (1)2
13. 2. USE (1)1

(1) QED BY (1)1, (1)2

If a Pcr has an extension, applying PriorPcr to it yields a Pcr.

**THEOREM** ThmPcrPriorIsPcr ∆

\( \forall p \in Pcr : PcrHasExtension(p) \Rightarrow PcrPrior(p) \in Pcr \)

**PROOF**

1. TAKE \( p \in Pcr \)
2. HAVE \( PcrHasExtension(p) \)
(1) USE DEF PcrHasExtension
(1) USE DEF PcrPrior
(1) USE DEF PcrLen
(1) USE DEF Pcr
(1) PcrPrior(p).extq ∈ Seq(Pcrx)
  (2) Len(p.extq) ∈ Nat BY ThmSeqLenIsNat
  (2) Len(p.extq) − 1 ∈ Nat BY ThmNatDecZero
  (2) QED BY ThmNatLess
(1) QED OBVIOUS

Putting the last extension back on the prior pcr yields the original pcr.

THEOREM ThmPcrExtendPriorLast ≜
∀ p ∈ Pcr :
PcrHasExtension(p) ⇒
PcrExtend(PcrPrior(p), PcrLastExtension(p)) = p

PROOF
(1) TAKE p ∈ Pcr
(1) HAVE PcrHasExtension(p)
(1) USE DEF PcrHasExtension
(1) USE DEF PcrPrior
(1) USE DEF PcrLastExtension
(1) USE DEF PcrExtend
(1) USE DEF PcrLen
(1) USE DEF Pcr
(1) DEFINE p0 ≜ PcrPrior(p)
(1) DEFINE x ≜ PcrLastExtension(p)
(1) PcrExtend(p0, x).init = p.init
OBVIOUS
(1) PcrExtend(p0, x).extq = p.extq
  (2) DEFINE qp ≜ p.extq
(2) SUFFICES
  ASSUME
    Len(qp) > 0,
    qp ∈ Seq(Pcrx)
  PROVE
    Append(SubSeq(qp, 1, Len(qp) − 1), qp[Len(qp)]) = qp
OBVIOUS
(2) HIDE DEF qp
(2) QED BY ThmSeqAppendPriorLast
(1) QED BY ThmPcrEqual
WELL KNOWN PCR VALUES

Value of the application pcr attained by rebooting.

THEOREM ThmAppRebootIsPcr $\triangleq$ AppReboot $\in$ Pcr

PROOF
(1) USE DEF AppReboot
(1) USE DEF Pcr
(1) QED BY ThmPcInitIsPcr

Value of the secure execution mode pcr attained by rebooting.

THEOREM ThmSemRebootIsPcr $\triangleq$ SemReboot $\in$ Pcr

PROOF
(1) USE DEF SemReboot
(1) USE DEF Pcr
(1) QED BY ThmPcInitIsPcr

Value of the secure execution mode pcr attained by entering the protected module in secure execution mode.

THEOREM ThmSemProtectIsPcr $\triangleq$ SemProtect $\in$ Pcr

PROOF
(1) USE DEF SemProtect
(1) USE DEF Pcr
(1) QED BY ThmPcInitIsPcr

Value of the secure execution mode pcr that indicates that Pasture is happy. Recovery has been properly performed and bound keys may be used. Checkpoint has not yet been invoked.

THEOREM ThmSemHappyIsPcr $\triangleq$ SemHappy $\in$ Pcr

PROOF
(1) USE DEF SemHappy
(1) USE ThmSemProtectIsPcr
(1) USE DEF Pcr
(1) QED BY ThmPcExtendIsPcr

From SemReboot cannot reach SemProtect.

THEOREM ThmSemRebootNotleqSemProtect $\triangleq$ $\neg$PcrLeq(SemReboot, SemProtect)

PROOF
(1) USE DEF PcrInit
(1) USE DEF SemReboot
(1) Use DEF SemProtect
(1) Use AssSemProtect
(1) QED by DEF PcrLeq

PROTECTED NV RAM STATE

THEOREM ThmInitNsNsNv ≜ InitNv ∈ Nv

PROOF
(1) Use DEF InitNv
(1) Use DEF Nv
(1) Use ThmAppRebootIsPcr
(1) QED obvious

SEAL OPERATION TRANSPORT SESSION STATE

THEOREM ThmNullTsNsTs ≜ NullTs ∈ Ts

PROOF
(1) Use DEF Ts
(1) QED obvious

THEOREM ThmNullTsNsSignedTs ≜ NullTs /∈ SignedTs

PROOF
(1) Use DEF NullTs
(1) Use NoSetContainsEverything
(1) QED obvious
PROOF OF INVARIANT InvType

It holds in the initial state.

THEOREM ThmInitInvType \[\triangleq\] Init \[\Rightarrow\] InvType

PROOF
(1) HAVE Init
(1) USE DEF Init
(1) USE DEF Pc, PcRecover, PcCheck
(1) USE DEF PcIdle
(1) USE DEF PcRECOV1, PcRECOV2, PcRECOV3
(1) USE DEF PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5

Just walk through each variable.

(1) InvType!nv BY ThmInitNvIsNv
(1) InvType!appPcr BY ThmAppRebootIsPcr
(1) InvType!semPcr BY ThmSemRebootIsPcr
(1) InvType!bootCtr OBVIOUS
(1) InvType!pc OBVIOUS
(1) InvType!chkpts BY ThmNullTIsTs
(1) InvType!tsvalues OBVIOUS
(1) InvType!obtains OBVIOUS
(1) InvType!revokes OBVIOUS
(1) QED BY DEF InvType

If it holds in the current state, and we perform a Next action, then it will hold in the next state.

Note that none of the Bug* definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

THEOREM ThmNextInvType \[\triangleq\] InvType \[\land\] [Next]vars \[\Rightarrow\] InvType' 

PROOF
(1) HAVE InvType \[\land\] [Next]vars
(1) USE DEF InvType
(1) USE DEF Pc, PcRecover, PcCheck
(1) USE DEF PcIdle
(1) USE DEF PcRECOV1, PcRECOV2, PcRECOV3
(1) USE DEF PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5

Say \textit{QED} here so that the rest of the proof has an indentation level. This creates a place where I can use the user interface renumbering operation to renumber all of the alternatives below.

(1) QED

Stutter step.

\langle 2 \rangle. 1. \textsc{CASE} vars' = vars
\langle 3 \rangle. USE \langle 2 \rangle.1
Walk through all Next alternatives.

(2) 2. CASE NextObtainAccess
    (3) USE NextObtainAccess
    (3) USE DEF NextObtainAccess
    (3) QED OBVIOUS

(2) 3. CASE NextProveRevoke
    (3) USE NextProveRevoke
    (3) USE DEF NextProveRevoke
    (3) PcrPrior(appPcr) ∈ Pcr BY ThmPcrPriorIsPcr
    (3) QED BY ThmPcrExtendIsPcr DEF Pcrx

(2) 4. CASE NextReboot
    (3) USE NextReboot
    (3) USE DEF NextReboot
    (3) InvType!appPcr' BY ThmAppRebootIsPcr
    (3) InvType!semPcr' BY ThmSemRebootIsPcr
    (3) InvType!chkptts' BY ThmNullTsIsTs
    (3) QED OBVIOUS

(2) 5. CASE NextForgetSealTs
    (3) USE NextForgetSealTs
    (3) USE DEF NextForgetSealTs
    (3) QED OBVIOUS

(2) 6. CASE Next ExtendAppPcr
    (3) USE NextExtendAppPcr
    (3) USE DEF NextExtendAppPcr
    (3) appPcr' ∈ Pcr BY ThmPcrExtendIsPcr DEF Pcrx
    (3) QED OBVIOUS

(2) 7. CASE NextExtendSemPcr
    (3) USE NextExtendSemPcr
    (3) USE DEF NextExtendSemPcr
    (3) InvType!semPcr' BY ThmPcrExtendIsPcr DEF Pcrx
    (3) QED OBVIOUS

(2) 9. CASE NextIncBootCtr
    (3) USE NextIncBootCtr
    (3) USE DEF NextIncBootCtr
    (3) bootCtr' ∈ Nat
        (4) 1. bootCtr + 1 ∈ Nat BY SMT
        (4) 2. bootCtr' = bootCtr + 1 OBVIOUS
        (4) QED BY {4}1, {4}2
    (3) QED OBVIOUS
\( \langle 2 \rangle \) 10. CASE NextEnterSemRecov
\( \langle 3 \rangle \) USE NextEnterSemRecov
\( \langle 3 \rangle \) USE DEF NextEnterSemRecov
\( \langle 3 \rangle \) InvType \( ! \) semPcr' BY ThmSemProtectIsPcr
\( \langle 3 \rangle \) InvType ! pc' OBVIOUS
\( \langle 3 \rangle \) InvType ! chkpts' OBVIOUS
\( \langle 3 \rangle \) QED OBVIOUS

\( \langle 2 \rangle \) 11. CASE NextSemRecov1 WhenCorrect
\( \langle 3 \rangle \) USE NextSemRecov1 WhenCorrect
\( \langle 3 \rangle \) USE DEF NextSemRecov1 WhenCorrect
\( \langle 3 \rangle \) QED OBVIOUS

\( \langle 2 \rangle \) 12. CASE NextSemRecov1 WhenIncorrect
\( \langle 3 \rangle \) USE NextSemRecov1 WhenIncorrect
\( \langle 3 \rangle \) USE DEF NextSemRecov1 WhenIncorrect
\( \langle 3 \rangle \) semPcr' \( \in \) Pcr BY ThmPcrExtendIsPcr
\( \langle 3 \rangle \) DEF Pcrx
\( \langle 3 \rangle \) QED OBVIOUS

\( \langle 2 \rangle \) 13. CASE NextSemRecov2
\( \langle 3 \rangle \) USE NextSemRecov2
\( \langle 3 \rangle \) USE DEF NextSemRecov2
\( \langle 3 \rangle \) nv' \( \in \) Nv
\( \langle 4 \rangle \) 1. nv \( \in \) Nv OBVIOUS
\( \langle 4 \rangle \) 2. nv'.current \( \in \) BOOLEAN BY DEF Nv
\( \langle 4 \rangle \) 3. nv' = [nv EXCEPT !.current = nv'.current] OBVIOUS
\( \langle 4 \rangle \) QED BY ONLY \( \langle 4 \rangle \) 1, \( \langle 4 \rangle \) 2, \( \langle 4 \rangle \) 3 DEF Nv
\( \langle 3 \rangle \) QED OBVIOUS

\( \langle 2 \rangle \) 14. CASE NextSemRecov3
\( \langle 3 \rangle \) USE NextSemRecov3
\( \langle 3 \rangle \) USE DEF NextSemRecov3
\( \langle 3 \rangle \) semPcr' \( \in \) Pcr BY ThmPcrExtendIsPcr
\( \langle 3 \rangle \) DEF Pcrx
\( \langle 3 \rangle \) pc' \( \in \) Pc BY DEF Pc, PcRecov
\( \langle 3 \rangle \) QED OBVIOUS

\( \langle 2 \rangle \) 15. CASE NextSealTs
\( \langle 3 \rangle \) USE NextSealTs
\( \langle 3 \rangle \) USE DEF NextSealTs
\( \langle 3 \rangle \) semPcr' \( \in \) Pcr BY ThmPcrExtendIsPcr
\( \langle 3 \rangle \) tsvalues' \( \in \) SUBSET Ts
\( \langle 4 \rangle \) 1. tsvalues \( \in \) SUBSET Ts OBVIOUS
\( \langle 4 \rangle \) DEFINE ts \( \triangleq \) NextSealTs! : !ts
\( \langle 4 \rangle \) 2. ts \( \in \) Ts BY DEF Ts, SignedTs
\( \langle 4 \rangle \) 3. tsvalues' = tsvalues \( \cup \) \{ ts \} OBVIOUS
\( \langle 4 \rangle \) QED BY \( \langle 4 \rangle \) 1, \( \langle 4 \rangle \) 2, \( \langle 4 \rangle \) 3
\( \langle 3 \rangle \) QED OBVIOUS
\(\langle 2.16.\) Case NextEnterSemChkpt
\(\langle 3.\) Use NextEnterSemChkpt
\(\langle 3.\) Use Def NextEnterSemChkpt
\(\langle 3.\) semPcr' ∈ Pcr By ThmSemProtectIsPcr
\(\langle 3.\) QED Obvious

\(\langle 2.17.\) Case NextSemChkpt1 WhenCorrect
\(\langle 3.\) Use NextSemChkpt1 WhenCorrect
\(\langle 3.\) Use Def NextSemChkpt1 WhenCorrect
\(\langle 3.\) chkptts' ∈ SignedTs
\(\langle 4.\) Use Def EnterSemChkptPredicate
\(\langle 4.\) Use Def Ts
\(\langle 4.\) QED Obvious
\(\langle 3.\) QED Obvious

\(\langle 2.18.\) Case NextSemChkpt1 WhenIncorrect
\(\langle 3.\) Use NextSemChkpt1 WhenIncorrect
\(\langle 3.\) Use Def NextSemChkpt1 WhenIncorrect
\(\langle 3.\) semPcr' ∈ Pcr By ThmPcrExtendIsPcr Def Pcrx
\(\langle 3.\) QED Obvious

\(\langle 2.19.\) Case NextSemChkpt2
\(\langle 3.\) Use NextSemChkpt2
\(\langle 3.\) Use Def NextSemChkpt2
\(\langle 3.\) nv' ∈ Nv
\(\langle 4.\) Define nvappPcr1 = NextSemChkpt2! : !nvappPcr1
\(\langle 4.\) 1. nvappPcr1 ∈ Pcr By Def SignedTs
\(\langle 4.\) QED By \(\langle 4.\) 1 Def Nv
\(\langle 3.\) QED Obvious

\(\langle 2.20.\) Case NextSemChkpt3
\(\langle 3.\) Use NextSemChkpt3
\(\langle 3.\) Use Def NextSemChkpt3
\(\langle 3.\) bootCtr' ∈ Nat
\(\langle 4.\) 1. bootCtr ∈ Nat Obvious
\(\langle 4.\) 2. bootCtr + 1 ∈ Nat By Only \(\langle 4.\) 1, SMT
\(\langle 4.\) QED By \(\langle 4.\) 1, \(\langle 4.\) 2
\(\langle 3.\) QED Obvious

\(\langle 2.21.\) Case NextSemChkpt4
\(\langle 3.\) Use NextSemChkpt4
\(\langle 3.\) Use Def NextSemChkpt4
\(\langle 3.\) nv' ∈ Nv By Def Nv
\(\langle 3.\) QED Obvious

\(\langle 2.22.\) Case NextSemChkpt5
\(\langle 3.\) Use NextSemChkpt5
\(\langle 3.\) Use Def NextSemChkpt5
\[ \langle 3 \rangle \text{semPer} \in \text{Per} \text{BY ThmPerExtendIsPer} \text{DEF Perx} \]
\[ \langle 3 \rangle \text{QED OBVIOUS} \]

\[ \langle 2 \rangle \text{QED} \]
\[ \text{BY} \langle 2 \rangle 1, \]
\[ \langle 2 \rangle 2, \langle 2 \rangle 3, \langle 2 \rangle 4, \langle 2 \rangle 5, \langle 2 \rangle 6, \langle 2 \rangle 7, \langle 2 \rangle 8, \langle 2 \rangle 9, \langle 2 \rangle 10, \]
\[ \langle 2 \rangle 11, \langle 2 \rangle 12, \langle 2 \rangle 13, \langle 2 \rangle 14, \langle 2 \rangle 15, \langle 2 \rangle 16, \langle 2 \rangle 17, \langle 2 \rangle 18, \]
\[ \langle 2 \rangle 19, \langle 2 \rangle 20, \langle 2 \rangle 21, \langle 2 \rangle 22 \]
\[ \text{DEF Next} \]

It is an invariant of the specification.

**THEOREM** \( \text{ThmInvType} \overset{\Delta}{=} \)
\[ \text{Spec} \Rightarrow \Box \text{InvType} \]

**PROOF**
\[ \langle 1 \rangle \text{Init} \Rightarrow \text{InvType} \text{BY ThmInitInvType} \]
\[ \langle 1 \rangle \text{InvType} \land [\text{Next}]_{\text{vars}} \Rightarrow \text{InvType}' \text{BY ThmNextInvType} \]
\[ \langle 1 \rangle \text{QED} \]

**PROOF OF INVARIANT** \( \text{InvInSemProtect} \)

It holds in the initial state.

**THEOREM** \( \text{ThmInitInvInSemProtect} \overset{\Delta}{=} \)
\[ \text{Init} \Rightarrow \text{InvInSemProtect} \]

**PROOF**
\[ \langle 1 \rangle \text{HAVE Init} \]
\[ \langle 1 \rangle \text{InvType} \text{BY ThmInitInvType} \]
\[ \langle 1 \rangle \text{USE DEF Init} \]
\[ \langle 1 \rangle \text{USE DEF InSem} \]
\[ \langle 1 \rangle \text{QED BY DEF InvInSemProtect} \]

If it holds in the current state, and we perform a \textit{Next} action, then it will hold in the next state.

Note that none of the Bug* definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

**THEOREM** \( \text{ThmNextInvInSemProtect} \overset{\Delta}{=} \)
\[ \text{InvInSemProtect} \land [\text{Next}]_{\text{vars}} \Rightarrow \text{InvInSemProtect}' \]
PROOF

(1) HAVE InvInSemProtect $\land$ $[\text{Next}\text{]}_{\text{vars}}$
(1) USE DEF InvInSemProtect
(1) USE DEF InSem

(1) InvType' BY ThmNextInvType
(1) InvInSemProtect! goal'

(2) USE DEF PcIDLE
(2) USE DEF PcRECOV1, PcRECOV2, PcRECOV3
(2) USE DEF PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5

Stutter step.

(2)1. CASE vars' = vars
   (3) USE (2)1
   (3) USE DEF vars
   (3) QED OBVIOUS

Walk through all Next alternatives.

(2)2. CASE NextObtainAccess
   (3) USE NextObtainAccess
   (3) USE DEF NextObtainAccess
   (3) QED OBVIOUS

(2)3. CASE NextProveRevoke
   (3) USE NextProveRevoke
   (3) USE DEF NextProveRevoke
   (3) QED OBVIOUS

(2)4. CASE NextReboot
   (3) USE NextReboot
   (3) USE DEF NextReboot
   (3) USE DEF PcrLeq
   (3) USE DEF SemProtect
   (3) USE DEF SemReboot
   (3) USE DEF PcrInit
   (3) USE DEF Pcri
   (3) USE AssSemProtect
   (3) QED OBVIOUS

(2)5. CASE NextForgetSealTs
   (3) USE NextForgetSealTs
   (3) USE DEF NextForgetSealTs
   (3) QED OBVIOUS

(2)6. CASE NextExtendAppPcr
   (3) USE NextExtendAppPcr
   (3) USE DEF NextExtendAppPcr
(3) QED OBVIOUS

(2) 7. CASE NextExtendSemPcr
    (3) USE NextExtendSemPcr
    (3) USE DEF NextExtendSemPcr
    (3) QED OBVIOUS

(2) 9. CASE NextIncBootCtr
    (3) USE NextIncBootCtr
    (3) USE DEF NextIncBootCtr
    (3) QED OBVIOUS

(2) 10. CASE NextEnterSemRecov
    (3) USE NextEnterSemRecov
    (3) USE DEF NextEnterSemRecov
    (3) QED OBVIOUS

(2) 11. CASE NextSemRecov1 WhenCorrect
    (3) USE NextSemRecov1 WhenCorrect
    (3) USE DEF NextSemRecov1 WhenCorrect
    (3) QED OBVIOUS

(2) 12. CASE NextSemRecov1 WhenIncorrect
    (3) USE NextSemRecov1 WhenIncorrect
    (3) USE DEF NextSemRecov1 WhenIncorrect
    (3) USE DEF PcrExtend
    (3) QED OBVIOUS

(2) 13. CASE NextSemRecov2
    (3) USE NextSemRecov2
    (3) USE DEF NextSemRecov2
    (3) QED OBVIOUS

(2) 14. CASE NextSemRecov3
    (3) USE NextSemRecov3
    (3) USE DEF NextSemRecov3
    (3) USE DEF PcrExtend
    (3) QED OBVIOUS

(2) 15. CASE NextSealTs
    (3) USE NextSealTs
    (3) USE DEF NextSealTs
    (3) QED OBVIOUS

(2) 16. CASE NextEnterSemchkpt
    (3) USE NextEnterSemchkpt
    (3) USE DEF NextEnterSemchkpt
    (3) QED OBVIOUS
\( (2) \) 17. CASE NextSemChkpt1 WhenCorrect \\
(3) USE NextSemChkpt1 WhenCorrect \\
(3) USE DEF NextSemChkpt1 WhenCorrect \\
(3) QED OBVIOUS \\

\( (2) \) 18. CASE NextSemChkpt1 WhenIncorrect \\
(3) USE NextSemChkpt1 WhenIncorrect \\
(3) USE DEF NextSemChkpt1 WhenIncorrect \\
(3) USE DEF PcrExtend \\
(3) QED OBVIOUS \\

\( (2) \) 19. CASE NextSemChkpt2 \\
(3) USE NextSemChkpt2 \\
(3) USE DEF NextSemChkpt2 \\
(3) QED OBVIOUS \\

\( (2) \) 20. CASE NextSemChkpt3 \\
(3) USE NextSemChkpt3 \\
(3) USE DEF NextSemChkpt3 \\
(3) QED OBVIOUS \\

\( (2) \) 21. CASE NextSemChkpt4 \\
(3) USE NextSemChkpt4 \\
(3) USE DEF NextSemChkpt4 \\
(3) QED OBVIOUS \\

\( (2) \) 22. CASE NextSemChkpt5 \\
(3) USE NextSemChkpt5 \\
(3) USE DEF NextSemChkpt5 \\
(3) USE DEF PcrExtend \\
(3) QED OBVIOUS \\

\( (2) \) QED \\
BY 1, \\
(2)2, (2)3, (2)4, (2)5, (2)6, (2)7, (2)9, (2)10, \\
(2)11, (2)12, (2)13, (2)14, (2)15, (2)16, (2)17, (2)18, \\
(2)19, (2)20, (2)21, (2)22 \\
DEF Next \\
(1) QED OBVIOUS \\

It is an invariant of the specification.

THEOREM ThmInvInSemProtect \( \triangleq \) \\
\( Spec \Rightarrow \square \text{InvInSemProtect} \) \\
PROOF \\
(1) Init \Rightarrow InvInSemProtect BY ThmInitInvInSemProtect \\
(1) InvInSemProtect \land [Next]_{vars} \Rightarrow InvInSemProtect'
PROOF OF INVARIANT InvUnreachableSemProtect

It holds in the initial state.

THEOREM ThmInitInvUnreachableSemProtect \( \triangleq \)
\( \text{Init} \Rightarrow \text{InvUnreachableSemProtect} \)

PROOF
(1) HAVE Init
(1) InvType BY ThmInitInvType
(1) InvInSemProtec\( \text{BY ThmInitInvSemProtect} \)
(1) USE DEF Init
(1) USE DEF InSem
(1) \( \neg \text{InSem} \) OBVIOUS
(1) \( \neg \text{PcrLeq}(\text{semPcr, SemProtect}) \)
\( \langle 2 \rangle \) USE DEF PcrLeq
\( \langle 2 \rangle \) USE DEF SemProtect
\( \langle 2 \rangle \) USE DEF SemReboot
\( \langle 2 \rangle \) USE DEF PcrInit
\( \langle 2 \rangle \) USE DEF Pcri
\( \langle 2 \rangle \) QED BY AssSemProtect
(1) QED BY DEF InvUnreachableSemProtect

If it holds in the current state, and we perform a Next action, then it will hold in the next state.

Note that none of the Bug* definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

THEOREM ThmNextInvUnreachableSemProtect \( \triangleq \)
\( \text{InvUnreachableSemProtect} \land [\text{Next}]_\text{vars} \Rightarrow \text{InvUnreachableSemProtect}' \)

PROOF
(1) HAVE InvUnreachableSemProtect \( \land [\text{Next}]_\text{vars} \)
(1) USE DEF InvUnreachableSemProtect
(1) USE DEF InSem
(1) USE DEF InvInSemProtect
\begin{enumerate}
  \item \textit{InvType}' \textbf{by} ThmNextInvType
  \item \textit{InvInSemProtect}' \textbf{by} ThmNextInvInSemProtect
  \item \textit{InvUnreachableSemProtect} \textbf{!} goal'
  \end{enumerate}

\begin{enumerate}
  \item USE \textit{DEF} \textit{PcIDLE}
  \item USE \textit{DEF} \textit{PcRECOV1}, \textit{PcRECOV2}, \textit{PcRECOV3}
  \item USE \textit{DEF} \textit{PcCHKPT1}, \textit{PcCHKPT2}, \textit{PcCHKPT3}, \textit{PcCHKPT4}, \textit{PcCHKPT5}
\end{enumerate}

Stutter step.
\begin{enumerate}
  \item CASE \textit{vars}' = \textit{vars}
    \item USE \textit{DEF} \textit{vars}
    \item QED OBVIOUS
\end{enumerate}

Walk through all \textit{Next} alternatives.
\begin{enumerate}
  \item CASE \textit{NextObtainAccess}
    \item USE \textit{DEF} \textit{NextObtainAccess}
    \item QED OBVIOUS
  \item CASE \textit{NextProveRevoke}
    \item USE \textit{DEF} \textit{NextProveRevoke}
    \item QED OBVIOUS
  \item CASE \textit{NextReboot}
    \item USE \textit{DEF} \textit{NextReboot}
    \item USE \textit{DEF} \textit{PcrLeq}
    \item USE \textit{DEF} \textit{SemProtect}
    \item USE \textit{DEF} \textit{SemReboot}
    \item USE \textit{DEF} \textit{PcrInit}
    \item USE \textit{DEF} \textit{Pcri}
    \item USE \textit{DEF} \textit{AssSemProtect}
    \item QED OBVIOUS
  \item CASE \textit{NextForgetSealTs}
    \item USE \textit{DEF} \textit{NextForgetSealTs}
    \item QED OBVIOUS
  \item CASE \textit{NextExtendAppPcr}
    \item USE \textit{DEF} \textit{NextExtendAppPcr}
    \item QED OBVIOUS
  \item CASE \textit{NextExtendSemPcr}
    \item USE \textit{DEF} \textit{NextExtendSemPcr}
\end{enumerate}
\(\text{(3)}\)  USE DEF \(\text{NextExtendSemPcr}\)
\(\text{(3)}\)  USE DEF \(\text{InvType}\)
\(\text{(3)}\)  \(\neg \text{PcrLeq}(\text{semPcr}', \text{SemProtect})\)
\(\text{(4)}\)  \(\text{SemProtect} \in \text{Pcr}\) BY \(\text{ThmSemProtectIsPcr}\)
\(\text{(4)}\)  QED BY \(\text{ThmPcrExtendSourceUnreachable}\)
\(\text{(3)}\)  QED OBVIOUS

\(\text{(2.9). CASE NextIncBootCtr}\)
\(\text{(3)}\)  USE \(\text{NextIncBootCtr}\)
\(\text{(3)}\)  USE DEF \(\text{NextIncBootCtr}\)
\(\text{(3)}\)  QED OBVIOUS

\(\text{(2.10). CASE NextEnterSemRecov}\)
\(\text{(3)}\)  USE \(\text{NextEnterSemRecov}\)
\(\text{(3)}\)  USE DEF \(\text{NextEnterSemRecov}\)
\(\text{(3)}\)  QED OBVIOUS

\(\text{(2.11). CASE NextSemRecov1 WhenCorrect}\)
\(\text{(3)}\)  USE \(\text{NextSemRecov1 WhenCorrect}\)
\(\text{(3)}\)  USE DEF \(\text{NextSemRecov1 WhenCorrect}\)
\(\text{(3)}\)  QED OBVIOUS

\(\text{(2.12). CASE NextSemRecov1 WhenIncorrect}\)
\(\text{(3)}\)  USE \(\text{NextSemRecov1 WhenIncorrect}\)
\(\text{(3)}\)  USE DEF \(\text{NextSemRecov1 WhenIncorrect}\)
\(\text{(3)}\)  USE DEF \(\text{InvType}\)
\(\text{(3)}\)  USE DEF \(\text{Pcrx}\)
\(\text{(3)}\)  \(\neg \text{PcrLeq}(\text{semPcr}', \text{SemProtect})\)
\(\text{(4)}\)  QED BY \(\text{ThmPcrExtendSelfUnreachable}\)
\(\text{(3)}\)  QED OBVIOUS

\(\text{(2.13). CASE NextSemRecov2}\)
\(\text{(3)}\)  USE \(\text{NextSemRecov2}\)
\(\text{(3)}\)  USE DEF \(\text{NextSemRecov2}\)
\(\text{(3)}\)  QED OBVIOUS

\(\text{(2.14). CASE NextSemRecov3}\)
\(\text{(3)}\)  USE \(\text{NextSemRecov3}\)
\(\text{(3)}\)  USE DEF \(\text{NextSemRecov3}\)
\(\text{(3)}\)  USE DEF \(\text{InvType}\)
\(\text{(3)}\)  USE DEF \(\text{Pcrx}\)
\(\text{(3)}\)  \(\neg \text{PcrLeq}(\text{semPcr}', \text{SemProtect})\)
\(\text{(4)}\)  QED BY \(\text{ThmPcrExtendSelfUnreachable}\)
\(\text{(3)}\)  QED OBVIOUS

\(\text{(2.15). CASE NextSealTs}\)
\(\text{(3)}\)  USE \(\text{NextSealTs}\)
\(\text{(3)}\)  USE DEF \(\text{NextSealTs}\)
(3) USE DEF \textit{InvType}

OPT: Reworked proof here to deal with extension of \textit{SEM PCR}

(3) CASE \textit{BugSealNoExt} OBVIOUS
(3) CASE $\neg \textit{BugSealNoExt}$

(4) $\neg \text{PcrLeq}(\text{semPcr}', \text{SemProtect})$

(5) $\text{SemProtect} \in \text{Pcr}$ BY ThmSemProtectIsPcr

(5) QED BY ThmPcrExtendSourceUnreachable DEF \textit{Pcrx}

(4) QED OBVIOUS
(3) QED OBVIOUS

(2) 16. CASE NextEnterSemChkpt
(3) USE NextEnterSemChkpt
(3) USE DEF NextEnterSemChkpt
(3) QED OBVIOUS

(2) 17. CASE NextSemChkpt1\text{WhenCorrect}
(3) USE NextSemChkpt1\text{WhenCorrect}
(3) USE DEF NextSemChkpt1\text{WhenCorrect}
(3) QED OBVIOUS

(2) 18. CASE NextSemChkpt1\text{WhenIncorrect}
(3) USE NextSemChkpt1\text{WhenIncorrect}
(3) USE DEF NextSemChkpt1\text{WhenIncorrect}
(3) USE DEF \textit{InvType}
(3) USE DEF \textit{Pcrx}

(3) $\neg \text{PcrLeq}(\text{semPcr}', \text{SemProtect})$

(4) QED BY ThmPcrExtendSelfUnreachable

(3) QED OBVIOUS

(2) 19. CASE NextSemChkpt2
(3) USE NextSemChkpt2
(3) USE DEF NextSemChkpt2
(3) QED OBVIOUS

(2) 20. CASE NextSemChkpt3
(3) USE NextSemChkpt3
(3) USE DEF NextSemChkpt3
(3) QED OBVIOUS

(2) 21. CASE NextSemChkpt4
(3) USE NextSemChkpt4
(3) USE DEF NextSemChkpt4
(3) QED OBVIOUS

(2) 22. CASE NextSemChkpt5
(3) USE NextSemChkpt5
(3) USE DEF NextSemChkpt5
(3) USE DEF InvType
(3) USE DEF Pcrx
(3) ~PcrLeq(semPcr', SemProtect)
(4) QED BY ThmPcrExtendSelfUnreachable
(3) QED OBVIOUS

(2) QED
BY (2)1,
(2)2, (2)3, (2)4, (2)5, (2)6, (2)7, (2)9, (2)10,
(2)11, (2)12, (2)13, (2)14, (2)15, (2)16, (2)17, (2)18,
(2)19, (2)20, (2)21, (2)22
DEF Next
(1) QED OBVIOUS

It is an invariant of the specification.

THEOREM ThmUnreachableSemProtect
 Spec ⇒ □ InvUnreachableSemProtect
PROOF
(1) Init ⇒ InvUnreachableSemProtect BY ThmInitInvUnreachableSemProtect
(1) InvUnreachableSemProtect ∧ [Next]vars ⇒ InvUnreachableSemProtect' BY ThmNextInvUnreachableSemProtect
(1) USE DEF Spec
(1) QED

PROOF OF INVARIANT InvNvProtection

It is an invariant of the specification.

THEOREM ThmInvNvProtection
 Spec ⇒ □ InvNvProtection
PROOF
(1) InvInSemProtect ∧ InvUnreachableSemProtect ⇒ InvNvProtection
(2) HAVE InvInSemProtect ∧ InvUnreachableSemProtect
PROOF OF OPTIMIZED PASTURE

152

PROOF OF INVARIANT InvSignedTsLeqBoot

It holds in the initial state.

THEOREM ThmInitInvSignedTsLeqBoot \triangleq

\hspace{1em} Init \Rightarrow InvSignedTsLeqBoot

PROOF

(1) HAVE Init
(1) InvType \hspace{1em} ThmInitInvType
(1) USE DEF Init
(1) USE DEF InvSignedTsLeqBoot
(1) InvSignedTsLeqBoot \hspace{1em} goal
(2) TAKE ts \in tsvalues \cup \{chkptts\}
If it holds in the current state, and we perform a $\text{Next}$ action, then it will hold in the next state.

Note that none of the Bug$^*$ definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

**THEOREM** $\text{ThmNextInvSignedTsLeqBoot} \triangleq \text{InvSignedTsLeqBoot} \land [\text{Next}]_{\text{vars}} \Rightarrow \text{InvSignedTsLeqBoot}'$

**PROOF**

(1) **HAVE** $\text{InvSignedTsLeqBoot} \land [\text{Next}]_{\text{vars}}$

(1) **USE DEF** $\text{InvSignedTsLeqBoot}$

(1) $\text{InvType'}$ **BY** $\text{ThmNextInvType}$

(1) $\text{InvSignedTsLeqBoot}!\text{goal'}$

<table>
<thead>
<tr>
<th>Stutter step.</th>
</tr>
</thead>
</table>
| (2) **CASE** $\text{vars'} = \text{vars}$
| (3) **USE** (2) I
| (3) **USE DEF** $\text{vars}$
| (3) **QED OBVIOUS** |

<table>
<thead>
<tr>
<th>Walk through all $\text{Next}$ alternatives.</th>
</tr>
</thead>
</table>
| (2) **CASE** $\text{NextObtainAccess}$
| (3) **USE** $\text{NextObtainAccess}$
| (3) **USE DEF** $\text{NextObtainAccess}$
| (3) **QED OBVIOUS** |

| (2) **CASE** $\text{NextProveRevoke}$
| (3) **USE** $\text{NextProveRevoke}$
| (3) **USE DEF** $\text{NextProveRevoke}$
| (3) **QED OBVIOUS** |

| (2) **CASE** $\text{NextReboot}$
| (3) **USE** $\text{NextReboot}$
| (3) **USE DEF** $\text{NextReboot}$
| (3) **QED BY** $\text{ThmNullTsIsntSignedTs}$ |

| (2) **CASE** $\text{NextForgetSealTs}$
| (3) **USE** $\text{NextForgetSealTs}$
| (3) **USE DEF** $\text{NextForgetSealTs}$
| (3) **QED OBVIOUS** |

| (2) **CASE** $\text{NextExtendAppPcr}$
| (3) **USE** $\text{NextExtendAppPcr}$
| (3) **USE DEF** $\text{NextExtendAppPcr}$
| (3) **QED OBVIOUS** |
(2.7) CASE NextExtendSemPer
  (3) USE NextExtendSemPer
  (3) USE DEF NextExtendSemPer
  (3) QED OBVIOUS

(2.9) CASE NextIncBootCtr
  (3) USE NextIncBootCtr
  (3) USE DEF NextIncBootCtr
  (3) USE DEF InvType
  (3) USE DEF SignedTs
  (3) bootCtr ≤ bootCtr + 1 BY ThmNatMore
  (3) QED BY ThmNatLeqIsTransitive

(2.10) CASE NextEnterSemRecov
  (3) USE NextEnterSemRecov
  (3) USE DEF NextEnterSemRecov
  (3) QED OBVIOUS

(2.11) CASE NextSemRecov1 WhenCorrect
  (3) USE NextSemRecov1 WhenCorrect
  (3) USE DEF NextSemRecov1 WhenCorrect
  (3) QED OBVIOUS

(2.12) CASE NextSemRecov1 WhenIncorrect
  (3) USE NextSemRecov1 WhenIncorrect
  (3) USE DEF NextSemRecov1 WhenIncorrect
  (3) QED OBVIOUS

(2.13) CASE NextSemRecov2
  (3) USE NextSemRecov2
  (3) USE DEF NextSemRecov2
  (3) QED OBVIOUS

(2.14) CASE NextSemRecov3
  (3) USE NextSemRecov3
  (3) USE DEF NextSemRecov3
  (3) QED OBVIOUS

(2.15) CASE NextSealTs
  (3) USE NextSealTs
  (3) USE DEF NextSealTs
  (3) DEFINE ts \(=\) NextSealTs! : !ts
  (3) ts.bootCtr ≤ bootCtr
      (4) USE DEF InvType
      (4) USE ThmNatLeqIsReflexive
      (4) QED OBVIOUS
  (3) QED OBVIOUS
16. **CASE** `NextEnterSemChkpt`
   (3) **USE** `NextEnterSemChkpt`
   (3) **USE DEF** `NextEnterSemChkpt`
   (3) **QED OBVIOUS**

17. **CASE** `NextSemChkpt1 WhenCorrect`
   (3) **USE** `NextSemChkpt1 WhenCorrect`
   (3) **USE DEF** `NextSemChkpt1 WhenCorrect`
   (3) **QED OBVIOUS**

18. **CASE** `NextSemChkpt1 WhenIncorrect`
   (3) **USE** `NextSemChkpt1 WhenIncorrect`
   (3) **USE DEF** `NextSemChkpt1 WhenIncorrect`
   (3) **QED OBVIOUS**

19. **CASE** `NextSemChkpt2`
   (3) **USE** `NextSemChkpt2`
   (3) **USE DEF** `NextSemChkpt2`
   (3) **QED OBVIOUS**

20. **CASE** `NextSemChkpt3`
   (3) **USE** `NextSemChkpt3`
   (3) **USE DEF** `NextSemChkpt3`
   (3) **USE DEF** `InvType`
   (3) **USE** `SignedTs`
   (3) `bootCtr ≤ bootCtr'`
   (4) **USE** `ThmNatLegIsReflexive`
   (4) **USE** `ThmNatMore`
   (4) **QED OBVIOUS**
   (3) **USE** `ThmNatLegIsTransitive`
   (3) **QED OBVIOUS**

21. **CASE** `NextSemChkpt4`
   (3) **USE** `NextSemChkpt4`
   (3) **USE DEF** `NextSemChkpt4`
   (3) **QED OBVIOUS**

22. **CASE** `NextSemChkpt5`
   (3) **USE** `NextSemChkpt5`
   (3) **USE DEF** `NextSemChkpt5`
   (3) **QED OBVIOUS**

(2) **QED**

BY (2) 1,
(2) 2, (2) 3, (2) 4, (2) 5, (2) 6, (2) 7, (2) 9, (2) 10, (2) 11, (2) 12, (2) 13, (2) 14, (2) 15, (2) 16, (2) 17, (2) 18, (2) 19, (2) 20, (2) 21, (2) 22

**DEF** `Next`
PROOF OF INVARIANT \textit{InvUnforgeableSemHappy}

It holds in the initial state.

\textbf{THEOREM} ThmInitInvUnforgeableSemHappy $\triangleq$
\[ \text{Init} \Rightarrow \text{InvUnforgeableSemHappy} \]
\textbf{PROOF}
\begin{enumerate}
\item HAVE Init
\item \textit{InvType} BY ThmInitInvType
\item \textit{InvInSemProtect} BY ThmInitInvInSemProtect
\item USE DEF Init
\item USE DEF \textit{InSem}
\item \textit{¬InSem} OBVIOUS
\item \textit{¬PcrLeq}(semPcr, SemHappy)
\item USE DEF PcrLeq
\item USE DEF PcrInit
\item USE DEF PcrExtend
\item USE DEF SemHappy
\item USE DEF SemReboot
\item USE DEF SemProtect
\item USE AssSemProtect
\item QED OBVIOUS
\item QED BY DEF \textit{InvUnforgeableSemHappy}
\end{enumerate}

If it holds in the current state, and we perform a \textit{Next} action, then it will hold in the next state.

Note that none of the Bug* definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

\textbf{THEOREM} ThmNextInvUnforgeableSemHappy $\triangleq$
\[ \text{InvUnforgeableSemHappy} \land [\text{Next}]_{vars} \Rightarrow \text{InvUnforgeableSemHappy'} \]
\textbf{PROOF}
\begin{enumerate}
\item HAVE InvUnforgeableSemHappy $\land [\text{Next}]_{vars}$
\item USE DEF InvUnforgeableSemHappy
\item USE DEF \textit{InSem}
1. \( \text{InvType}' \) BY \( \text{ThmNextInvType} \)
2. \( \text{InvInSemProtect}' \) BY \( \text{ThmNextInvInSemProtect} \)
3. \( \text{InvUnforgeableSemHappy} \) goal'

- USE DEF \( \text{PcIDLE} \)
- USE DEF \( \text{PcRECOV1, PcRECOV2, PcRECOV3} \)
- USE DEF \( \text{PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5} \)

Stutter step.

1. CASE \( \text{vars}' = \text{vars} \)
   - USE \( \langle 2 \rangle 1 \)
   - USE DEF \( \text{vars} \)
   - QED OBVIOUS

Walk through all \( \text{Next} \) alternatives.

2. CASE \( \text{NextObtainAccess} \)
   - USE \( \text{NextObtainAccess} \)
   - USE DEF \( \text{NextObtainAccess} \)
   - QED OBVIOUS

3. CASE \( \text{NextProveRevoke} \)
   - USE \( \text{NextProveRevoke} \)
   - USE DEF \( \text{NextProveRevoke} \)
   - QED OBVIOUS

4. CASE \( \text{NextReboot} \)
   - USE \( \text{NextReboot} \)
   - USE DEF \( \text{NextReboot} \)
   - USE DEF \( \text{PcrLeq} \)
   - USE DEF \( \text{PcrInit} \)
   - USE DEF \( \text{PcrExtend} \)
   - USE DEF \( \text{SemHappy} \)
   - USE DEF \( \text{SemReboot} \)
   - USE DEF \( \text{SemProtect} \)
   - USE \( \text{AssSemProtect} \)
   - QED OBVIOUS

5. CASE \( \text{NextForgetSealTs} \)
   - USE \( \text{NextForgetSealTs} \)
   - USE DEF \( \text{NextForgetSealTs} \)
   - QED OBVIOUS

6. CASE \( \text{NextExtendAppPcr} \)
   - USE \( \text{NextExtendAppPcr} \)
   - USE DEF \( \text{NextExtendAppPcr} \)
   - QED OBVIOUS

7. CASE \( \text{NextExtendSemPcr} \)
(3) USE NextExtendSemPcr
(3) USE DEF NextExtendSemPcr
(3) USE DEF InvType
(3) HAVE ¬InSem' 
(3) 1. CASE semPcr = SemHappy
    (4) USE (3)1
    (4) QED BY ThmPcrExtendSelfUnreachable
(3) 2. CASE ¬PcrLeq(semPcr, SemHappy)
    (4) USE (3)2
    (4) SemHappy ∈ Pcr BY ThmSemHappyIsPcr
    (4) ¬PcrLeq(semPcr', SemHappy) BY ThmPcrExtendSourceUnreachable
    (4) QED OBVIOUS
(3) QED BY (3)1, (3)2

(2) 9. CASE NextIncBootCtr
    (3) USE NextIncBootCtr
    (3) USE DEF NextIncBootCtr
    (3) QED OBVIOUS

(2) 10. CASE NextEnterSemRecov
    (3) USE NextEnterSemRecov
    (3) USE DEF NextEnterSemRecov
    (3) QED OBVIOUS

(2) 11. CASE NextSemRecov1 WhenCorrect
    (3) USE NextSemRecov1 WhenCorrect
    (3) USE DEF NextSemRecov1 WhenCorrect
    (3) QED OBVIOUS

(2) 12. CASE NextSemRecov1 WhenIncorrect
    (3) USE NextSemRecov1 WhenIncorrect
    (3) USE DEF NextSemRecov1 WhenIncorrect
    (3) USE DEF InvType
    (3) USE Pcrx
    (3) semPcr = SemProtect BY DEF InvInSemProtect
    (3) USE DEF SemHappy
    (3) USE AssSemHappy
    (3) USE ThmPcrExtendIncompatible
    (3) QED OBVIOUS

(2) 13. CASE NextSemRecov2
    (3) USE NextSemRecov2
    (3) USE DEF NextSemRecov2
    (3) QED OBVIOUS

(2) 14. CASE NextSemRecov3
    (3) USE NextSemRecov3
    (3) USE DEF NextSemRecov3
(3) $\text{semPcr} = \text{SemProtect}$ BY DEF $\text{InvInSemProtect}$
(3) USE DEF $\text{SemHappy}$
(3) QED OBVIOUS

(2) 15. CASE NextSealTs
(3) USE NextSealTs
(3) USE DEF NextSealTs
(3) USE DEF InvType

************************************************************
OPT: REMOVED PROOF HERE TO DEAL WITH EXTENSION OF SEM PCR
************************************************************

(3) CASE $\neg \text{BugSealNoExt}$ OBVIOUS
(3) CASE $\neg \neg \text{BugSealNoExt}$
   (4) HAVE $\neg \text{InSem'}$
   (4) 1. CASE $\text{semPcr} = \text{SemHappy}$
       (5) USE (4) 1
       (5) QED BY ThmPcrExtendSelfUnreachable DEF Pcrx
   (4) 2. CASE $\neg \text{PerLeq(} \text{semPcr, SemHappy)}$
       (5) USE (4) 2
       (5) $\text{SemHappy} \in \text{Pcr}$ BY ThmSemHappyIsPcr
       (5) $\neg \text{PerLeq(} \text{semPcr', SemHappy)}$ BY ThmPerExtendSourceUnreachable DEF Pcrx
       (5) QED OBVIOUS
   (4) QED BY (4) 1, (4) 2
(3) QED OBVIOUS

(2) 16. CASE NextEnterSemChkpt
(3) USE NextEnterSemChkpt
(3) USE DEF NextEnterSemChkpt
(3) QED OBVIOUS

(2) 17. CASE NextSemChkpt1 WhenCorrect
(3) USE NextSemChkpt1 WhenCorrect
(3) USE DEF NextSemChkpt1 WhenCorrect
(3) QED OBVIOUS

(2) 18. CASE NextSemChkpt1 WhenIncorrect
(3) USE NextSemChkpt1 WhenIncorrect
(3) USE DEF NextSemChkpt1 WhenIncorrect
(3) USE DEF InvType
(3) USE DEF Pcrx
(3) $\text{semPcr} = \text{SemProtect}$ BY DEF $\text{InvInSemProtect}$
(3) USE DEF $\text{SemHappy}$
(3) USE $\text{AssSemHappy}$
(3) USE ThmPcrExtendIncompatible
(3) QED OBVIOUS

(2) 19. CASE NextSemChkpt2
(3) USE NextSemChkpt2
It is an invariant of the specification.

**THEOREM** \( \text{ThmInvUnforgeableSemHappy} \mapsto \text{Spec} \Rightarrow \square \text{InvUnforgeableSemHappy} \)

**PROOF**

1. \( \text{Init} \Rightarrow \text{InvUnforgeableSemHappy} \) \( \text{by} \ \text{ThmInitInvUnforgeableSemHappy} \)
2. \( \text{InvUnforgeableSemHappy} \land [\text{Next}]_{\text{vars}} \Rightarrow \text{InvUnforgeableSemHappy}' \)

\( \text{by} \ \text{ThmNextInvUnforgeableSemHappy} \)

1. \( \text{use def Spec} \)
2. \( \text{qed} \)
PROOF OF INVARIANT InvProperLastExtension

It holds in the initial state.

**THEOREM** ThmInitInvProperLastExtension \( \Delta \)

\[ \text{Init} \Rightarrow \text{InvProperLastExtension} \]

**PROOF**

1. HAVE \text{Init}
2. \text{InvType} BY ThmInitInvType
3. QED BY DEF InvProperLastExtension, \text{Init}

If it holds in the current state, and we perform a Next action, then it will hold in the next state.

**THEOREM** ThmNextInvProperLastExtension \( \Delta \)

\[ \text{InvProperLastExtension} \land [\text{Next}]_{\text{vars}} \Rightarrow \text{InvProperLastExtension}' \]

**PROOF**

1. HAVE \text{InvProperLastExtension} \land [\text{Next}]_{\text{vars}}
2. USE DEF InvProperLastExtension
3. \text{InvType}' BY ThmNextInvType
4. \text{InvProperLastExtension}' \text{goal}'

Wow, this is an easy one.

1. USE DEF \text{vars}
2. USE DEF NextObtainAccess
3. USE DEF NextProveRevoke
4. USE DEF NextReboot
5. USE DEF NextForgetSealTs
6. USE DEF NextExtendAppPcr
7. USE DEF NextExtendSemPcr
8. USE DEF NextIncBootCtr
9. USE DEF NextEnterSemRecov
10. USE DEF NextSemRecov1 WhenCorrect
11. USE DEF NextSemRecov1 WhenIncorrect
12. USE DEF NextSemRecov2
13. USE DEF NextSemRecov3
14. USE DEF NextSealTs
15. USE DEF NextEnterSemChkpt
16. USE DEF NextSemChkpt1 WhenCorrect
17. USE DEF NextSemChkpt1 WhenIncorrect
18. USE DEF NextSemChkpt2
19. USE DEF NextSemChkpt3
20. USE DEF NextSemChkpt4
21. USE DEF NextSemChkpt5
\(\{2\}\) QED by Def \textit{Next}
\(\{1\}\) QED obvious

It is an invariant of the specification.

**THEOREM** ThmInvProperLastExtension \(\triangleq\)
\[Spec \Rightarrow \Box \text{InvProperLastExtension}\]

**PROOF**
\(\{1\}\) \textit{Init} \Rightarrow \text{InvProperLastExtension} \hspace{1em} \text{BY ThmInitInvProperLastExtension}
\(\{1\}\) \text{InvProperLastExtension} \land \text{\([Next]\_\textit{vars} \Rightarrow \text{InvProperLastExtension}'\)}
\hspace{1em} \text{BY ThmNextInvProperLastExtension}
\(\{1\}\) \text{USE DEF Spec}
\(\{1\}\) QED

---

**PROOF OF INVARIANT InvOneLog**

It holds in the initial state.

**THEOREM** ThmInitInvOneLog \(\triangleq\)
\[\text{Init} \Rightarrow \text{InvOneLog}\]

**PROOF**
\(\{1\}\) \text{HAVE Init}
\(\{1\}\) \text{InvType} \hspace{1em} \text{BY ThmInitInvType}
\(\{1\}\) \text{InvSignedTsLeqBoot} \hspace{1em} \text{BY ThmInitInvSignedTsLeqBoot}
\(\{1\}\) \text{InvInSemProtect} \hspace{1em} \text{BY ThmInitInvInSemProtect}
\(\{1\}\) \text{InvUnForgeableSemHappy} \hspace{1em} \text{BY ThmInitInvUnForgeableSemHappy}
\(\{1\}\) \text{InvProperLastExtension} \hspace{1em} \text{BY ThmInitInvProperLastExtension}
\(\{1\}\) \text{USE DEF Init}
\(\{1\}\) \text{USE DEF InitNv}
\(\{1\}\) \text{USE DEF PcIDLE}
\(\{1\}\) \text{USE DEF PcRECOV1, PcRECOV2, PcRECOV3}
\(\{1\}\) \text{USE DEF PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5}
\(\{1\}\) \text{LogInNv} \hspace{1em} \text{BY DEF LogInNv}
\(\{1\}\) \text{\texttt{¬LogInApp}}
\(\{2\}\) \text{semPcr} \neq \text{SemHappy}
\(\{3\}\) \text{semPcr.init} \neq \text{SemHappy.init}
(4) \textit{sempcr.init} \neq \textit{SemProtect.init}  
(5) USE DEF \textit{SemReboot}  
(5) USE DEF \textit{SemProtect}  
(5) USE DEF \textit{PcrInit}  
(5) QED BY \textit{AssSemProtect}  
(4) USE DEF \textit{SemHappy}  
(4) QED BY DEF \textit{PcrExtend}  
(3) QED BY DEF \textit{Pcr}  
(2) QED BY DEF \textit{LogInApp}  
(1) \neg \textit{LogInTs} \land \textit{AllCurrentTs} = \{\}  
(2) \neg \textit{CheckTsIsCurrent} (\textit{chkptts})  
(3) USE DEF \textit{CheckTsIsCurrent}  
(3) QED BY \textit{ThmNullTsIsnSignedTs}  
(2) USE DEF \textit{AllCurrentTs}  
(2) QED BY DEF \textit{LogInTs}  
(1) \textit{InvOneLog} ! \textit{goal} ! obtains  
(1) \textit{InvOneLog} ! \textit{goal} ! revokes  
(1) \textit{InvVerifiableRevocation}  
(1) QED BY DEF \textit{InvOneLog}  

If it holds in the current state, and we perform a \textit{Next} action, then it will hold in the next state.

\textbf{THEOREM} \textit{ThmNextInvOneLog}  
\textit{InvOneLog} \land [\textit{Next}]_{vars} \Rightarrow \textit{InvOneLog}'  

\textbf{PROOF}  
(1) HAVE \textit{InvOneLog} \land [\textit{Next}]_{vars}  
(1) \textit{InvType}  
(1) \textit{InvSignedTsLeqBoot}  
(1) \textit{InvInSemProtect}  
(1) \textit{InvUnforgeableSemHappy}  
(1) \textit{InvProperLastExtension}  
(1) \textit{InvType}'  
(1) \textit{InvSignedTsLeqBoot}'  
(1) \textit{InvInSemProtect}'  
(1) \textit{InvUnforgeableSemHappy}'  
(1) \textit{InvProperLastExtension}'  
(1) \textit{InvOneLog} ! \textit{goal}'  
(2) USE DEF \textit{PcIDLE}  
(2) USE DEF \textit{PcRECOV1}, \textit{PcRECOV2}, \textit{PcRECOV3}  
(2) USE DEF \textit{PcCHKPT1}, \textit{PcCHKPT2}, \textit{PcCHKPT3}, \textit{PcCHKPT4}, \textit{PcCHKPT5}
Stutter step.

1. Case \( \text{vars}' = \text{vars} \)

2. Use (2)

3. Use \( \text{vars} \)

4. Unchanged \( \text{CheckTsIsCurrent}(\text{chkptts}) \)

5. Use \( \text{AllCurrentTs} \)

6. Use \( \text{CheckTsIsCurrent} \)

7. QED by \( \text{LogInTs} \)

8. Unchanged \( \text{CurrentTsLog} \)

9. Use \( \text{LogInNv} \)

10. Unchanged \( \text{LogInApp} \)

11. Unchanged \( \text{LogInTs} \)

12. \( \text{InvOneLog} \)

13. \( \text{InvOneLog}' \)

14. QED by \( \text{InvOneLog} \)

NextObtainAccess or NextProveRevoke

2. Case \( \text{NextObtainAccess} \vee \text{NextProveRevoke} \)

3. Use (2)

4. Use \( \text{NextObtainAccess} \)

5. Use \( \text{NextProveRevoke} \)

6. Unchanged \( \text{CheckTsIsCurrent}(\text{chkptts}) \)

7. Use \( \text{AllCurrentTs} \)

8. Use \( \text{CheckTsIsCurrent} \)

9. QED by \( \text{LogInTs} \)

10. Unchanged \( \text{CurrentTsLog} \)

11. Unchanged \( \text{LogInNv} \)

12. Unchanged \( \text{LogInApp} \)

13. Unchanged \( \text{LogInTs} \)

14. \( \text{LogInApp} \)

15. \( \text{NextObtainAccess} \)

NextObtainAccess is predicated on the fact that the log is in the application pcr.

This depends on

\[ \wedge \text{BugObtainAccessNoCheckHappy} \triangleq \text{FALSE} \]

16. BugObtainAccessNoCheckHappy = FALSE by \( \text{BugObtainAccessNoCheckHappy} \)

17. QED by \( \text{LogInApp} \)

18. \( \text{NextProveRevoke} \)

NextProveRevoke is predicated on the fact that the log is in the application pcr.

This depends on

\[ \wedge \text{BugProveRevokeNoCheckHappy} \triangleq \text{FALSE} \]
(5) \textbf{BugProveRevokeNoCheckHappy} = \textit{FALSE} BY DEF BugProveRevokeNoCheckHappy

(5) QED BY DEF LogInApp

(4) QED OBVIOUS

(3) \texttt{InvOneLog \ 'goal!obtains' \ \& \ \texttt{InvOneLog \ '!goal!revokes' \}}

Since the log is in the application pcr, putting a copy of the application pcr into obtains or into revokes preserves the invariant that everything in obtains \cup revokes can reach the log.

(4) PcrLeq(appPcr, \ appPcr)BY ThmPcrLeqIsReflexive DEF InvType

(4) QED BY DEF IsOnLog, InvOneLog

(3) \texttt{InvVerifiableRevocation' \}}

Since we only add an element to obtains \cup revokes when the log is in the application pcr, we know that all elements in obtains \cup revokes in the new state must be on the log, which we can check as \leq app pcr.

So we proceed with proof by contradiction. Assuming that verifiable deletion will be violated in the new state, we pick the \( o \in \text{obtains} \) and \( r \in \text{revokes} \) whose \texttt{PcrPrior}'s are the same. But since both \( o \) and \( r \) must be \leq app pcr, this means that their last extension must be the same. This contradicts the assumption that \texttt{OBTAIN} is different from \texttt{REVOKE}.

(4) CASE \texttt{InvVerifiableRevocation' \ OBVIOUS}

(4) CASE \texttt{\neg InvVerifiableRevocation' \}}

(5) \textbf{PICK} \( o \in \text{obtains}' \), \( r \in \text{revokes}' : \text{PcrPrior}(o) = \text{PcrPrior}(r) \)

BY DEF InvVerifiableRevocation

(5) DEFINE \( p \triangleq \text{PcrPrior}(o) \)

(5) DEFINE \( xo \triangleq \text{PcrLastExtension}(o) \)

(5) DEFINE \( xr \triangleq \text{PcrLastExtension}(r) \)

(5) \( xo = \text{PcrxOBTAIN} \) BY DEF InvProperLastExtension

(5) \( xr = \text{PcrxREVOKE} \) BY DEF InvProperLastExtension

(5) \( o = \text{PcrExtend}(p, xo) \) BY ThmPcrExtendPriorLast DEF InvType, InvProperLastExtension

(5) \( r = \text{PcrExtend}(p, xr) \) BY ThmPcrExtendPriorLast DEF InvType, InvProperLastExtension

(5) PcrLeq(o, \ appPcr')BY DEF IsOnLog, InvOneLog

(5) PcrLeq(r, \ appPcr')BY DEF IsOnLog, InvOneLog

(5) \( xo \in \text{Pcrx} \) BY DEF Pcrx

(5) \( xr \in \text{Pcrx} \) BY DEF Pcrx

(5) \( xo = xr \)

The prover needs a lot of help to focus its attention.

(6) HIDE DEF \( p \)

(6) HIDE DEF \( xo \)

(6) HIDE DEF \( xr \)

(6) \( appPcr' \in \text{Pcr} \) BY DEF InvType

(6) QED BY ThmPcrExtendLeqAnticollision

(5) \( \text{PcrxOBTAIN} \neq \text{PcrxREVOKE} \) BY AssObtainNeqRevoke

(5) QED OBVIOUS

(4) QED OBVIOUS

(3) QED BY DEF InvOneLog

\texttt{NextReboot}
3. CASE NextReboot
(3) USE NextReboot
(3) USE DEF NextReboot
(3) UNCHANGED LogInNv BY DEF LogInNv
Cancels SemHappy if we had it, which erases any log that had been in the application pcr.

(3) ¬LogInApp
(4) semPcr′ ≠ SemHappy
   (5) semPcr′.init ≠ SemHappy.init
   (6) semPcr′.init ≠ SemProtect.init
   (7) USE DEF SemReboot
   (7) USE DEF SemProtect
   (7) USE DEF PcrInit
   (7) QED BY AssSemProtect
   (6) USE DEF SemHappy
   (6) QED BY DEF PcrExtend
   (5) QED BY DEF Pcr
(4) QED BY DEF LogInApp

Overwrites chkptts with an unsigned ts, which might erase a log that had been in the seal attestations.

(3) LogInTs′ ⇒ LogInTs
   (4) HAVE LogInTs′
   (4) AllCurrentTs′ ≠ ∅ BY DEF LogInTs
   (4) USE DEF AllCurrentTs
   (4) USE DEF CheckTsIsCurrent
   (4) ¬CheckTsIsCurrent(chkptts)′ BY ThmNullTsIsntSignedTs
   (4) ∃ ts ∈ tvalues′ : CheckTsIsCurrent(ts) OBVIOUS
   (4) QED BY DEF LogInTs

Any remaining current seal attestations existed previously, so they must contain the same log.

(3) ∀ ts1, ts2 ∈ AllCurrentTs′ : ts1.appPcr = ts2.appPcr
   (4) TAKE ts1, ts2 ∈ AllCurrentTs′
   (4) USE DEF AllCurrentTs
   (4) USE DEF CheckTsIsCurrent
   (4) ¬CheckTsIsCurrent(chkptts)′ BY ThmNullTsIsntSignedTs
   (4) ts1 ∈ AllCurrentTs OBVIOUS
   (4) ts2 ∈ AllCurrentTs OBVIOUS
   (4) QED BY DEF InvOneLog

If there are any remaining current seal attestations, the log in them has to be the same as before.

(3) LogInTs′ ⇒ UNCHANGED CurrentTsLog
   (4) HAVE LogInTs′
   (4) ∃ ts ∈ AllCurrentTs′ : ts ∈ AllCurrentTs
      (5) USE DEF LogInTs
      (5) USE DEF AllCurrentTs
      (5) USE DEF CheckTsIsCurrent
      (5) ¬CheckTsIsCurrent(chkptts)′ BY ThmNullTsIsntSignedTs
      (5) ∀ ts ∈ AllCurrentTs′ : ts ∈ AllCurrentTs OBVIOUS
\[\begin{align*}
\langle 5 \rangle & \text{ QED OBVIOUS} \\
\langle 4 \rangle & \forall t_1 \in \text{AllCurrentTs}' : \\
& \forall t_0 \in \text{AllCurrentTs} : \\
&t_1.\text{appPcr} = t_0.\text{appPcr} \\
& \text{BY DEF InvOneLog} \\
\langle 4 \rangle & \text{ QED BY DEF CurrentTsLog} \\
\langle 3 \rangle & \text{ InvOneLog'}\text{ obtains'} \quad \text{BY DEF IsOnLog, InvOneLog} \\
\langle 3 \rangle & \text{ InvOneLog'}\text{ revokes'} \quad \text{BY DEF IsOnLog, InvOneLog} \\
\langle 3 \rangle & \text{ InvVerifiableRevocation'} \quad \text{BY DEF InvVerifiableRevocation, InvOneLog} \\
\langle 3 \rangle & \text{ QED BY DEF InvOneLog} \\
\end{align*}\]

NextForgetSealTs

\[\begin{align*}
\langle 2 \rangle & 4. \text{ CASE NextForgetSealTs} \\
\langle 3 \rangle & \text{ USE NextForgetSealTs} \\
\langle 3 \rangle & \text{ USE DEF NextForgetSealTs} \\
\langle 3 \rangle & \text{ UNCHANGED LogInNv} \quad \text{BY DEF LogInNv} \\
\langle 3 \rangle & \text{ UNCHANGED LogInApp} \quad \text{BY DEF LogInApp} \\
\langle 3 \rangle & \text{ UNCHANGED CheckTsIsCurrent(chkptts)} \quad \text{BY DEF CheckTsIsCurrent} \\
\end{align*}\]

Forgets a seal attestation, which might erase a log that had been in the seal attestations.

\[\begin{align*}
\langle 3 \rangle & \text{ LogInTs'} \Rightarrow \text{ LogInTs} \\
\langle 4 \rangle & \text{ HAVE LogInTs'} \\
\langle 4 \rangle & \text{ AllCurrentTs'} \neq \{\} \quad \text{BY DEF LogInTs} \\
\langle 4 \rangle & \text{ USE DEF AllCurrentTs} \\
\langle 4 \rangle & \text{ USE DEF CheckTsIsCurrent} \\
\langle 4 \rangle & \text{ QED BY DEF LogInTs} \\
\end{align*}\]

Any remaining current seal attestations existed previously, so they must contain the same log.

\[\begin{align*}
\langle 3 \rangle & \forall t_1, t_2 \in \text{AllCurrentTs}' : t_1.\text{appPcr} = t_2.\text{appPcr} \\
\langle 4 \rangle & \text{ TAKE t_1, t_2 \in AllCurrentTs'} \\
\langle 4 \rangle & \text{ USE DEF AllCurrentTs} \\
\langle 4 \rangle & \text{ USE DEF CheckTsIsCurrent} \\
\langle 4 \rangle & \text{ ts_1 \in AllCurrentTs OBVIOUS} \\
\langle 4 \rangle & \text{ ts_2 \in AllCurrentTs OBVIOUS} \\
\langle 4 \rangle & \text{ QED BY DEF InvOneLog} \\
\end{align*}\]

If \(\text{chkptts}\) contains a current seal attestation, then the log is in the seal attestations.

\[\begin{align*}
\langle 3 \rangle & \text{ CheckTsIsCurrent(chkptts)'} \Rightarrow \text{ LogInTs'} \\
\langle 4 \rangle & \text{ USE DEF LogInTs} \\
\langle 4 \rangle & \text{ USE DEF AllCurrentTs} \\
\langle 4 \rangle & \text{ USE DEF CheckTsIsCurrent} \\
\langle 4 \rangle & \text{ QED OBVIOUS} \\
\end{align*}\]

If there are any remaining current seal attestations, the log in them has to be the same as before.

\[\begin{align*}
\langle 3 \rangle & \text{ LogInTs'} \Rightarrow \text{ UNCHANGED CurrentTsLog} \\
\langle 4 \rangle & \text{ HAVE LogInTs'} \\
\langle 4 \rangle & \exists t \in \text{AllCurrentTs'} : t \in \text{AllCurrentTs} \\
\end{align*}\]
\( \langle 5 \rangle \text{ USE DEF } \text{LogInTs} \)
\( \langle 5 \rangle \text{ USE DEF } \text{AllCurrentTs} \)
\( \langle 5 \rangle \text{ USE DEF } \text{CheckTsIsCurrent} \)
\( \langle 5 \rangle \forall ts \in \text{AllCurrentTs}': ts \in \text{AllCurrentTs}\text{OBVIOUS} \)
\( \langle 5 \rangle \text{ QED OBVIOUS} \)
\( \langle 4 \rangle \forall ts1 \in \text{AllCurrentTs}' : \)
\( \forall ts0 \in \text{AllCurrentTs} : \)
\( ts1.\text{appPcr} = ts0.\text{appPcr} \text{BY DEF } \text{InvOneLog} \)
\( \langle 4 \rangle \text{ QED BY DEF } \text{CurrentTsLog} \)
\( \langle 3 \rangle \text{ InvOneLog'}!\text{goal'obtains'} \text{BY DEF } \text{IsOnLog}, \text{InvOneLog} \)
\( \langle 3 \rangle \text{ InvOneLog'}!\text{goal'revokes'} \text{BY DEF } \text{IsOnLog}, \text{InvOneLog} \)
\( \langle 3 \rangle \text{ InvVerifiableRevocation'} \text{BY DEF } \text{InvVerifiableRevocation}, \text{InvOneLog} \)
\( \langle 3 \rangle \text{ QED BY DEF } \text{InvOneLog} \)

**NextExtendAppPcr**

\( \langle 5 \rangle \text{ USE DEF } \text{LogInTs} \)
\( \langle 5 \rangle \text{ USE DEF } \text{AllCurrentTs} \)
\( \langle 5 \rangle \text{ USE DEF } \text{CheckTsIsCurrent} \)
\( \langle 5 \rangle \forall ts \in \text{AllCurrentTs}': ts \in \text{AllCurrentTs}\text{OBVIOUS} \)
\( \langle 5 \rangle \text{ QED OBVIOUS} \)
\( \langle 4 \rangle \forall ts1 \in \text{AllCurrentTs}' : \)
\( \forall ts0 \in \text{AllCurrentTs} : \)
\( ts1.\text{appPcr} = ts0.\text{appPcr} \text{BY DEF } \text{LogInTs} \)
\( \langle 4 \rangle \text{ QED BY DEF } \text{CurrentTsLog} \)
\( \langle 3 \rangle \text{ InvOneLog'}!\text{goal'obtains'} \text{BY DEF } \text{IsOnLog}, \text{InvOneLog} \)
\( \langle 3 \rangle \text{ InvOneLog'}!\text{goal'revokes'} \text{BY DEF } \text{IsOnLog}, \text{InvOneLog} \)
\( \langle 3 \rangle \text{ InvVerifiableRevocation'} \text{BY DEF } \text{InvVerifiableRevocation}, \text{InvOneLog} \)
\( \langle 3 \rangle \text{ QED BY DEF } \text{InvOneLog} \)

CASE NextExtendAppPcr

\( \langle 3 \rangle \text{ USE } \text{NextExtendAppPcr} \)
\( \langle 3 \rangle \text{ USE DEF } \text{NextExtendAppPcr} \)
\( \langle 3 \rangle \text{ UNCHANGED } \text{CheckTsIsCurrent}(\text{chkptts})\text{BY DEF } \text{CheckTsIsCurrent} \)
\( \langle 3 \rangle \text{ UNCHANGED } \text{AllCurrentTs} \)
\( \langle 4 \rangle \text{ USE DEF } \text{AllCurrentTs} \)
\( \langle 4 \rangle \text{ USE DEF } \text{CheckTsIsCurrent} \)
\( \langle 4 \rangle \text{ QED BY DEF } \text{LogInTs} \)
\( \langle 3 \rangle \text{ UNCHANGED } \text{CurrentTsLog}\text{BY DEF } \text{CurrentTsLog} \)
\( \langle 3 \rangle \text{ UNCHANGED } \text{LogInNu} \text{BY DEF } \text{LogInNu} \)
\( \langle 3 \rangle \text{ UNCHANGED } \text{LogInApp} \text{BY DEF } \text{LogInApp} \)
\( \langle 3 \rangle \text{ UNCHANGED } \text{LogInTs} \)
\( \langle 4 \rangle \text{ USE DEF } \text{AllCurrentTs} \)
\( \langle 4 \rangle \text{ USE DEF } \text{CheckTsIsCurrent} \)
\( \langle 4 \rangle \text{ QED BY DEF } \text{LogInTs} \)
\( \langle 3 \rangle \text{ InvVerifiableRevocation'} \text{BY DEF } \text{InvVerifiableRevocation}, \text{InvOneLog} \)
\( \langle 3 \rangle \text{ InvOneLog'}!\text{goal'}obtains' \land \text{InvOneLog'}!\text{goal'}revokes' \)

CASE ¬LogInApp

\( \langle 5 \rangle \text{ USE DEF } \text{LogInApp} \)
\( \langle 5 \rangle \text{ USE DEF } \text{IsOnLog}, \text{InvOneLog} \)

**NextExtendAppPcr** is predicated on not being in sem, so none of the sem clauses apply.

\( \langle 5 \rangle \text{ USE DEF } \text{InSem} \)
\( \langle 5 \rangle \text{ QED OBVIOUS} \)

CASE LogInApp

\( \langle 5 \rangle \text{ USE DEF } \text{LogInApp} \)

If the log is in the application pcr, extending the application pcr preserves the fact that all entries in obtains \(\cup\) revokes can reach it.
∀ \ p \in \text{obtains} \cup \text{revokes} : \text{LogInApp} \Rightarrow \text{PcrLeq}(p, \ appPcr')

\text{UNCHANGED} (\text{obtains} \cup \text{revokes}) \text{OBVIOUS}

\text{TAK}e \ p \in \text{obtains} \cup \text{revokes}

\text{HAVE} \text{LogInApp}

\text{PcrLeq}(p, \ appPcr) \text{BY DEF IsOnLog, InvOneLog}

\text{PcrLeq}(appPcr, \ appPcr') \text{BY ThmPcrExtendLeq DEF InvType}

\text{QED BY ThmPcrLeqIsTransitive DEF InvType}

\text{QED BY DEF IsOnLog, InvOneLog}

\text{QED OBVIOUS}

\text{QED BY DEF InvOneLog}

NextExtendSemPcr

\text{CASE NextExtendSemPcr}

\text{USE NextExtendSemPcr}

\text{USE DEF NextExtendSemPcr}

\text{UNCHANGED CheckTsIsCurrent(chkptts) BY DEF CheckTsIsCurrent}

\text{UNCHANGED AllCurrentTs}

\text{USE DEF AllCurrentTs}

\text{USE DEF CheckTsIsCurrent}

\text{QED BY DEF LogInTs}

\text{UNCHANGED CurrentTsLog} \text{BY DEF CurrentTsLog}

\text{UNCHANGED LogInNv} \text{BY DEF LogInNv}

Cancels \text{SemHappy} if we had it, which erases any log that had been in the application pcr.

\text{UNCHANGED LogInApp'}

\text{UNCHANGED LogInTs}

\text{USE DEF AllCurrentTs}

\text{USE DEF CheckTsIsCurrent}

\text{QED BY DEF LogInTs}

\text{QED BY DEF LogInApp}

\text{QED BY DEF InvOneLog'}

\text{QED BY DEF InvOneLog'}

\text{QED BY DEF InvVerifiableRevocation', InvOneLog}

\text{QED BY DEF InvVerifiableRevocation, InvOneLog}

NextIncBootCtr
Since no signed \( ts \) seal can have a \( bootCtr \) greater than the current \( bootCtr \), incrementing \( bootCtr \) erases any log that had been in a seal attestation.

\[
\neg LogInTs'\]

\[
\forall ts \in tsvalues' \cup \{chkptts'\} : ts \in SignedTs \Rightarrow ts.bootCtr \neq bootCtr'
\]

\[
\{5\} \text{ HAVE } ts \in SignedTs
\]

\[
\{5\} ts.bootCtr \leq bootCtr' \text{ BY DEF } InvSignedTsLeqBoot
\]

\[
\{5\} bootCtr' \in Nat \text{ BY DEF } SignedTs
\]

\[
\{5\} bootCtr < bootCtr' \text{ BY ThmNatInc}
\]

\[
\{5\} ts.bootCtr < bootCtr' \text{ BY ThmNatLeqLt}
\]

\[
\{5\} \text{ QED BY ThmNatLeqXorGt, ThmNatLeqIsReflexive}
\]

\[
\{4\} \text{ QED BY DEF } LogInTs
\]

Erases all current \( ts \) seal attestations.

\[
\{3\} AllCurrentTs' = \{\} \land \neg CheckTsIsCurrent(chkptts')
\]

\[
\{4\} \text{ QED BY DEF } LogInTs
\]

NextEnterSemRecov

\[
\{2\} 9. \text{ CASE } NextEnterSemRecov
\]

\[
\{3\} \text{ USE } NextEnterSemRecov
\]

\[
\{3\} \text{ USE DEF } NextEnterSemRecov
\]

\[
\{3\} \text{ USE DEF } InSem
\]

\[
\{3\} \text{ UNCHANGED } CheckTsIsCurrent(chkptts) \text{ BY DEF } CheckTsIsCurrent
\]

\[
\{3\} \text{ UNCHANGED } AllCurrentTs
\]

\[
\{4\} \text{ USE DEF } AllCurrentTs
\]

\[
\{4\} \text{ QED BY DEF } LogInTs
\]

\[
\{3\} \text{ UNCHANGED } CurrentTsLog \text{ BY DEF } CurrentTsLog
\]
UNCHANGED \textit{LogInNv} \hspace{1cm} \textbf{BY DEF} \textit{LogInNv}

Cancels \textit{SemHappy} if we had it, which erases any log that had been in the application \textit{pcr}.

\(\neg \text{LogInApp}'\)

\(\text{semPer}' \neq \text{SemHappy}\)

\(\text{USE DEF} \quad \text{SemHappy}\)

\(\text{USE DEF} \quad \text{SemProtect}\)

\(\text{USE DEF} \quad \text{Pcri}\)

\(\text{USE DEF} \quad \text{Pcrx}\)

\(\text{USE ThmPcrInitIsPcr}\)

\(\text{USE ThmPcrExtendIsPcr}\)

\(\text{PcrLeq} (\text{SemProtect}, \text{SemHappy}) \hspace{1cm} \text{BY ThmPcrExtendLeq}\)

\(\neg \text{PcrLeq} (\text{SemHappy}, \text{SemProtect}) \hspace{1cm} \text{BY ThmPcrExtendSelfUnreachable}\)

\(\text{QED BY ThmPcrLeqIsAntisymmetric}\)

\(\text{QED BY DEF} \quad \text{LogInApp}\)

\(\text{UNCHANGED} \quad \text{LogInTs}\)

\(\text{USE DEF} \quad \text{AllCurrentTs}\)

\(\text{USE DEF} \quad \text{CheckTsIsCurrent}\)

\(\text{QED BY DEF} \quad \text{LogInTs}\)

\(\text{InvOneLog} \quad \text{goal} \quad \text{obtains}' \quad \text{BY DEF} \quad \text{IsOnLog}, \text{InvOneLog}\)

\(\text{InvOneLog} \quad \text{goal} \quad \text{revokes}' \quad \text{BY DEF} \quad \text{IsOnLog}, \text{InvOneLog}\)

\(\text{InvVerifiableRevocation}' \quad \text{BY DEF} \quad \text{InvVerifiableRevocation, InvOneLog}\)

\(\text{QED BY DEF} \quad \text{InvOneLog}\)

\textit{NextSemRecov1 WhenCorrect}

\(\text{CASE} \quad \text{NextSemRecov1 WhenCorrect}\)

\(\text{USE} \quad \text{NextSemRecov1 WhenCorrect}\)

\(\text{USE DEF} \quad \text{NextSemRecov1 WhenCorrect}\)

\(\text{UNCHANGED} \quad \text{CheckTsIsCurrent}(\text{chkptts}) \hspace{1cm} \text{BY DEF} \quad \text{CheckTsIsCurrent}\)

\(\text{UNCHANGED} \quad \text{AllCurrentTs}\)

\(\text{USE DEF} \quad \text{AllCurrentTs}\)

\(\text{USE DEF} \quad \text{CheckTsIsCurrent}\)

\(\text{QED BY DEF} \quad \text{LogInTs}\)

\(\text{UNCHANGED} \quad \text{CurrentTsLog} \quad \text{BY DEF} \quad \text{CurrentTsLog}\)

\(\text{UNCHANGED} \quad \text{LogInNv} \quad \text{BY DEF} \quad \text{LogInNv}\)

\(\text{UNCHANGED} \quad \text{LogInApp} \quad \text{BY DEF} \quad \text{LogInApp}\)

\(\text{UNCHANGED} \quad \text{LogInTs} \quad \text{BY DEF} \quad \text{LogInTs}\)

\textit{EnterSemRecovPredicate} guarantees that the log is in the \textit{nv} ram.

This depends on \(\text{BugRecovNoCheckCur} \neq \text{FALSE}\)

\(\text{LogInNv}\)

\(\text{USE DEF} \quad \text{EnterSemRecovPredicate}\)

\(\text{BugRecovNoCheckCur} = \text{FALSE} \hspace{1cm} \text{BY DEF} \quad \text{BugRecovNoCheckCur}\)

\(\text{QED BY DEF} \quad \text{LogInNv}\)
\textit{EnterSemRecovPredicate} guarantees that the application pcr equals the log saved in the \textit{nv} ram.

This depends on \textit{BugRecovNoCheckApp} \( = \text{FALSE} \)

\begin{align*}
(3) \quad \text{appPcr} &= \text{nv.appPcr} \\
(4) \quad \text{USE DEF \textit{EnterSemRecovPredicate}} \\
(4) \quad \text{BugRecovNoCheckApp} &= \text{FALSE} \quad \text{BY DEF \textit{BugRecovNoCheckApp}} \\
(4) \quad \text{QED OBITIOUS} \\
(3) \quad \text{InvOneLog'} \text{goal' obtains'} & \quad \text{BY DEF \textit{IsOnLog}, \textit{InvOneLog}} \\
(3) \quad \text{InvOneLog'} \text{goal' revokes'} & \quad \text{BY DEF \textit{IsOnLog}, \textit{InvOneLog}} \\
(3) \quad \text{InvVerifiableRevocation'} & \quad \text{BY DEF \textit{InvVerifiableRevocation}, \textit{InvOneLog}} \\
(3) \quad \text{QED BY DEF \textit{InvOneLog}}
\end{align*}

\begin{align*}
\text{NextSemRecov1WhenIncorrect} \\
(2) \quad \text{11. CASE NextSemRecov1WhenIncorrect} \\
(3) \quad \text{USE NextSemRecov1WhenIncorrect} \\
(3) \quad \text{USE DEF NextSemRecov1WhenIncorrect} \\
(3) \quad \text{UNCHANGED CheckTsIsCurrent(chkpts) BY DEF \textit{CheckTsIsCurrent}} \\
(3) \quad \text{UNCHANGED AllCurrentTs} \\
(4) \quad \text{USE DEF AllCurrentTs} \\
(4) \quad \text{USE DEF CheckTsIsCurrent} \\
(4) \quad \text{QED BY DEF LogInTs} \\
(3) \quad \text{UNCHANGED CurrentTsLog} & \quad \text{BY DEF \textit{CurrentTsLog}} \\
(3) \quad \text{UNCHANGED LogInNu} & \quad \text{BY DEF LogInNu}
\end{align*}

Extending \textit{sem pcr} with Unhappy results in something other than \textit{SemHappy}, which indicates that the log is not in the application pcr.

\begin{align*}
(3) \quad &\text{¬LogInApp'} \\
(4) \quad &\text{semPcr'} \neq \text{SemHappy} \\
(5) \quad &\text{USE DEF \textit{SemHappy}} \\
(5) \quad &\text{USE DEF \textit{SemProtect}} \\
(5) \quad &\text{USE DEF \textit{Pcri}} \\
(5) \quad &\text{USE DEF \textit{Pcrx}} \\
(5) \quad &\text{USE ThmPcrInitIsPcr} \\
(5) \quad &\text{USE ThmPcrExtendIsPcr} \\
(5) \quad &\text{semPcr} = \text{SemProtect} \quad \text{BY DEF \textit{InvInSemProtect}, \textit{InSem}} \\
(5) \quad &\text{USE AssSemHappy} \\
(5) \quad &\text{QED BY ThmPcrExtendAnticollision} \\
(4) \quad &\text{QED BY DEF LogInApp} \\
(3) \quad &\text{UNCHANGED LogInTs} \\
(4) \quad &\text{USE DEF AllCurrentTs} \\
(4) \quad &\text{USE DEF CheckTsIsCurrent} \\
(4) \quad &\text{QED BY DEF LogInTs} \\
(3) \quad &\text{InvOneLog'} \text{goal' obtains'} & \quad \text{BY DEF \textit{IsOnLog}, \textit{InvOneLog}} \\
(3) \quad &\text{InvOneLog'} \text{goal' revokes'} & \quad \text{BY DEF \textit{IsOnLog}, \textit{InvOneLog}} \\
(3) \quad &\text{InvVerifiableRevocation'} & \quad \text{BY DEF \textit{InvVerifiableRevocation}, \textit{InvOneLog}} \\
(3) \quad &\text{QED BY DEF \textit{InvOneLog}}
\end{align*}
NextSemRecov\textsuperscript{2}

\begin{enumerate}
\item \textbf{Case NextSemRecov\textsuperscript{2}}
\item \textbf{Use NextSemRecov\textsuperscript{2}}
\item \textbf{Use Def NextSemRecov\textsuperscript{2}}
\item \textbf{Unchanged CheckTsIsCurrent(chkptts) Def CheckTsIsCurrent}
\item \textbf{Unchanged AllCurrentTs}
\item \textbf{Use Def AllCurrentTs}
\item \textbf{Use Def CheckTsIsCurrent}
\item \textbf{Qed By Def LogInTs}
\item \textbf{Unchanged CurrentTsLog By Def CurrentTsLog}
\end{enumerate}

Clearing \texttt{nv} current erases the log from the \texttt{nv} ram.

This depends on \texttt{BugRecovNoClrCur} \equiv \texttt{false}

\begin{enumerate}
\item \texttt{¬LogInNv'}
\item \texttt{¬nv'.current}
\item \texttt{BugRecovNoClrCur = false By Def BugRecovNoClrCur}
\item \textbf{Qed By Def InvType, Nv}
\item \textbf{Qed By Def LogInNv}
\item \textbf{Unchanged LogInApp By Def LogInApp}
\item \textbf{Unchanged LogInTs By Def LogInTs}
\item \textbf{InvOneLog'} goal obtains' By Def IsOnLog, InvOneLog
\item \textbf{InvOneLog'} goal revokes' By Def IsOnLog, InvOneLog
\item \textbf{InvVerifiableRevocation'} By Def InvVerifiableRevocation, InvOneLog
\item \textbf{Qed By Def InvOneLog}
\end{enumerate}

NextSemRecov\textsuperscript{3}

\begin{enumerate}
\item \textbf{Case NextSemRecov\textsuperscript{3}}
\item \textbf{Use NextSemRecov3}
\item \textbf{Use Def NextSemRecov\textsuperscript{3}}
\item \textbf{Unchanged CheckTsIsCurrent(chkptts) Def CheckTsIsCurrent}
\item \textbf{Unchanged AllCurrentTs}
\item \textbf{Use Def AllCurrentTs}
\item \textbf{Use Def CheckTsIsCurrent}
\item \textbf{Qed By Def LogInTs}
\item \textbf{Unchanged CurrentTsLog By Def CurrentTsLog}
\item \textbf{Unchanged LogInNv By Def LogInNv}
\end{enumerate}

Extending sem pcr to \texttt{SemHappy} puts the log in the application pcr, provided that the seal pcr contains \texttt{SealReboot}.

But in the current state the log has no domicile. So the fact that its domicile might be the application pcr in the next state does not require a proof that it is not living anywhere else, since we get that for free.

\begin{enumerate}
\item \texttt{LogInApp'} ∈ BOOLEAN By Def LogInApp
\item \textbf{Unchanged LogInTs By Def LogInTs}
\item \textbf{InvOneLog'} goal obtains' By Def IsOnLog, InvOneLog
\item \textbf{InvOneLog'} goal revokes' By Def IsOnLog, InvOneLog
\item \textbf{InvVerifiableRevocation'} By Def InvVerifiableRevocation, InvOneLog
\end{enumerate}
QED BY DEF InvOneLog

NextSealTs

14. CASE NextSealTs

USE NextSealTs

USE DEF NextSealTs

UNCHANGED CheckTsIsCurrent(chkpts) BY DEF CheckTsIsCurrent

UNCHANGED LogInNu BY DEF LogInNu

Cancels SemHappy if we had it, which erases any log that had been in the application pcr.

This depends on BugSealNoExt $\Delta = \text{FALSE}$

---

**OPT: REWORKED PROOF HERE TO DEAL WITH EXTENSION OF SEM PCR**

---

semPcr$' \neq$ SemHappy

semPcr = SemHappy $\lor$ $\neg$PcrLeq(semPcr, SemHappy)

BY DEF InvUnforgeableSemHappy

semPcr $\in$ Pcr BY DEF InvType

SemHappy $\in$ Pcr BY ThmSemHappyIsPcr

$\neg$PcrLeq(semPcr, SemHappy)

semPcr$' = \text{PcrExtend}(\text{semPcr}, \text{PcrxSEAL})$

QED OBVIOUS

QED BY ThmPcrExtendFromEqOrNotleq DEF Pcrx

QED BY DEF LogInApp

---

InvVerifiableRevocation$'$ BY DEF InvVerifiableRevocation, InvOneLog

If the log was not in the application pcr, the resulting seal attestation will not be valid, so there is no change in AllCurrentTs or LogSummaryInTs.

---

CASE $\neg$LogInApp

UNCHANGED AllCurrentTs

USE DEF AllCurrentTs

USE DEF CheckTsIsCurrent

Use PICK to make ts a CONSTANT so that $\neg$CheckTsIsCurrent(ts)$'$ means the ts picked now evaluated with CheckTsIsCurrent in the next state.

PICK ts $\in$ SignedTs : ts = NextSealTs! : !ts BY DEF InvType, SignedTs

$\forall$ ts1 $\in$ tsvalues' : CheckTsIsCurrent(ts1)$'$ $\Rightarrow$ ts1 $\in$ tsvalues

tsvalues' = tsvalues $\cup$ \{ts\} OBVIOUS

$\neg$CheckTsIsCurrent(ts)$'$ BY DEF LogInApp

QED OBVIOUS

QED OBVIOUS

UNCHANGED LogInTs BY DEF LogInTs

UNCHANGED CurrentTsLog BY DEF CurrentTsLog

InvOneLog! goal! obtains' BY DEF IsOnLog, InvOneLog
If the log was in the application pcr, the resulting seal attestation will be valid. But then the old $\text{AllCurrentTs}$ had to be empty, since the log could not have been in the seal attestations.

(3) **CASE LogInApp**

Use **PICK** to make $ts$ a **CONSTANT** so that $\text{CheckTsIsCurrent}(ts)$' means the $ts$ picked now evaluated with $\text{CheckTsIsCurrent}$ in the next state.

(4) **PICK** $ts \in \text{SignedTs}$ : $ts = \text{NextSealTs}$! : $ts$ BY **InvType**, $\text{SignedTs}$

(5) $\text{CheckTsIsCurrent}(ts)$' BY **DEF** $\text{CheckTsIsCurrent}$, $\text{LogInApp}$

(5) **QED** BY **DEF** $\text{CheckTsIsCurrent}$

(4) $\text{AllCurrentTs}' = \{ts\}$

(5) $\forall ts1 \in tsvalues \cup \{\text{chkptts}'\} : \neg \text{CheckTsIsCurrent}(ts1)$'

(6) $\text{AllCurrentTs} = \{\}$

(7) $\neg \text{LogInTs}$ BY **DEF** $\text{InvOneLog}$

(7) **QED** BY **DEF** $\text{LogInTs}$

(6) $\neg \text{CheckTsIsCurrent}(\text{chkptts})'$

(7) $\neg \text{CheckTsIsCurrent}(\text{chkptts})$ BY **DEF** $\text{AllCurrentTs}$

(7) **QED** BY **DEF** $\text{CheckTsIsCurrent}$

(6) $\forall ts1 \in tsvalues : \neg \text{CheckTsIsCurrent}(ts1)$'

(7) $\forall ts1 \in tsvalues : \neg \text{CheckTsIsCurrent}(ts1)$ BY **DEF** $\text{AllCurrentTs}$

(7) **QED** BY **DEF** $\text{CheckTsIsCurrent}$

(6) **QED** BY **DEF** $\text{CheckTsIsCurrent}$

(5) $tsvalues' = tsvalues \cup \{ts\}$ **OBVIOUS**

(5) $ts \in \text{AllCurrentTs}'$ BY **DEF** $\text{AllCurrentTs}$

(5) **QED** BY **DEF** $\text{AllCurrentTs}$

(4) $\forall ts1, ts2 \in \text{AllCurrentTs}' : ts1.appPcr = ts2.appPcr$ **OBVIOUS**

(4) $\text{LogInTs}'$ BY **DEF** $\text{LogInTs}$

(4) $\text{CurrentTsLog}' = ts.appPcr$ BY **DEF** $\text{CurrentTsLog}$

(4) $\text{InvOneLog}! \text{goal}! \text{obtains}'$ BY **DEF** $\text{IsOnLog}$, $\text{InvOneLog}$

(4) $\text{InvOneLog}! \text{goal}! \text{revokes}'$ BY **DEF** $\text{IsOnLog}$, $\text{InvOneLog}$

(4) **QED** BY **DEF** $\text{InvOneLog}$

(3) **QED** **OBVIOUS**

---

**NextEnterSemChkpt**

(2) **15. CASE NextEnterSemChkpt**

(3) **USE NextEnterSemChkpt**

(3) **USE DEF NextEnterSemChkpt**

(3) **USE DEF InSem**

(3) **UNCHANGED LogInNv** BY **DEF** $\text{LogInNv}$

Cancels $\text{SemHappy}$ if we had it, so erases any log that might have been in the application pcr.

(3) $\neg \text{LogInApp}'$

(4) $\text{semPcr}' \neq \text{SemHappy}$

(5) **USE DEF SemHappy**
Overwrites $\text{chkptts}$ with a value from $\text{tsvalues}$, so if $\text{chkptts}$ had been the only seal log, we just erased it.

(3) $\text{LogInTs}' \Rightarrow \text{LogInTs}$
(4) $\text{HAVE LogInTs}'$
(4) $\text{AllCurrentTs}' \neq \{\}$ BY DEF $\text{LogInTs}$
(4) USE DEF $\text{AllCurrentTs}$
(4) USE DEF $\text{CheckTsIsCurrent}$
(4) $\text{chkptts}' \in \text{tsvalues}'$ OBVIOUS
(4) $\exists t \in \text{tsvalues}' : \text{CheckTsIsCurrent}(t)$ OBVIOUS
(4) QED BY DEF $\text{LogInTs}$

Any remaining current seal attestations existed previously, so they must contain the same log.

(3) $\forall ts_1, ts_2 \in \text{AllCurrentTs}' : ts_1.appPcr = ts_2.appPcr$
(4) TAKE $ts_1, ts_2 \in \text{AllCurrentTs}'$
(4) USE DEF $\text{AllCurrentTs}$
(4) USE DEF $\text{CheckTsIsCurrent}$
(4) $\text{chkptts}' \in \text{tsvalues}'$ OBVIOUS
(4) $ts_1 \in \text{AllCurrentTs}'$ OBVIOUS
(4) $ts_2 \in \text{AllCurrentTs}'$ OBVIOUS
(4) QED BY DEF $\text{InvOneLog}$

If there are any remaining current seal attestations, the log in them has to be the same as before.

(3) $\text{LogInTs}' \Rightarrow \text{UNCHANGED CurrentTsLog}$
(4) $\text{HAVE LogInTs}'$
(4) $\exists t \in \text{AllCurrentTs}' : t \in \text{AllCurrentTs}$
(5) USE DEF $\text{LogInTs}$
(5) USE DEF $\text{AllCurrentTs}$
(5) USE DEF $\text{CheckTsIsCurrent}$
(5) $\forall t \in \text{AllCurrentTs}' : t \in \text{AllCurrentTs}'$ OBVIOUS
(5) QED OBVIOUS
(4) $\forall ts_1 \in \text{AllCurrentTs}' :$
$\forall ts_0 \in \text{AllCurrentTs} :$
$ts_1.appPcr = ts_0.appPcr$
BY DEF $\text{InvOneLog}$
(4) QED BY DEF $\text{CurrentTsLog}$

(3) $\text{InvOneLog}'$ goal' obtains' BY DEF $\text{IsOnLog}', \text{InvOneLog}$
(3) $\text{InvOneLog}'$ goal' revokes' BY DEF $\text{IsOnLog}', \text{InvOneLog}$
(3) $\text{InvVerifiableRevocation}'$ BY DEF $\text{InvVerifiableRevocation}', \text{InvOneLog}$
(3) QED BY DEF $\text{InvOneLog}$

QED BY DEF $\text{LogInApp}$
NextSemChkpt1WhenCorrect

(2) 16. CASE NextSemChkpt1WhenCorrect

(3) USE NextSemChkpt1WhenCorrect

(3) USE DEF NextSemChkpt1WhenCorrect

(3) UNCHANGED CheckTsIsCurrent(chkptts) BY DEF CheckTsIsCurrent

(3) UNCHANGED AllCurrentTs

(4) USE DEF AllCurrentTs

(4) USE DEF CheckTsIsCurrent

(4) QED BY DEF LogInTs

(3) UNCHANGED CurrentTsLog BY DEF CurrentTsLog

(3) UNCHANGED LogInNv BY DEF LogInNv

(3) UNCHANGED LogInApp BY DEF LogInApp

(3) UNCHANGED LogInTs BY DEF LogInTs

EnterSemChkptPredicate guarantees that the log is in the seal attestations (in particular, in chkptts).

This depends on

\[ \land \ BugChkptNoCheckTsHappy \land \ BugChkptNoCheckTsCtr \equiv \text{FALSE} \]

\[ \land \ BugChkptNoCheckTsHappy \land \ BugChkptNoCheckTsCtr \equiv \text{FALSE} \]

\[ \land \ BugChkptNoCheckTsHappy = \text{FALSE} \]

\[ \land \ BugChkptNoCheckTsCtr = \text{FALSE} \]

\[ \land \ BugChkptNoCheckTsHappy = \text{FALSE} \]

\[ \land \ BugChkptNoCheckTsCtr = \text{FALSE} \]

\[ \land \ BugChkptNoCheckTsHappy = \text{FALSE} \]

\[ \land \ BugChkptNoCheckTsCtr = \text{FALSE} \]

\[ \land \ BugChkptNoCheckTsHappy = \text{FALSE} \]

\[ \land \ BugChkptNoCheckTsCtr = \text{FALSE} \]

\[ \land \ BugChkptNoCheckTsHappy = \text{FALSE} \]

\[ \land \ BugChkptNoCheckTsCtr = \text{FALSE} \]

(3) LogInTs \land CheckTsIsCurrent(chkptts) \land CurrentTsLog = chkptts.appPcr

(4) USE DEF EnterSemChkptPredicate

(4) BugChkptNoCheckTsHappy = \text{FALSE} BY DEF BugChkptNoCheckTsHappy

(4) BugChkptNoCheckTsCtr = \text{FALSE} BY DEF BugChkptNoCheckTsCtr

(4) CheckTsIsCurrent(chkptts) BY DEF CheckTsIsCurrent

(4) AllCurrentTs \neq \{\} BY DEF AllCurrentTs

(4) CurrentTsLog = chkptts.appPcr

(5) \forall ts \in AllCurrentTs : ts.appPcr = chkptts.appPcr

(5) QED BY DEF CurrentTsLog

(4) QED BY DEF LogInTs

(3) InvOneLog ! goal! obtains’ BY DEF IsOnLog, InvOneLog

(3) InvOneLog ! goal! revokes’ BY DEF IsOnLog, InvOneLog

(3) InvVerifiableRevocation’ BY DEF InvVerifiableRevocation, InvOneLog

(3) QED BY DEF InvOneLog

NextSemChkpt1WhenIncorrect

(2) 17. CASE NextSemChkpt1WhenIncorrect

(3) USE NextSemChkpt1WhenIncorrect

(3) USE DEF NextSemChkpt1WhenIncorrect

(3) UNCHANGED CheckTsIsCurrent(chkptts) BY DEF CheckTsIsCurrent

(3) UNCHANGED AllCurrentTs

(4) USE DEF AllCurrentTs

(4) USE DEF CheckTsIsCurrent

(4) QED BY DEF LogInTs

(3) UNCHANGED CurrentTsLog BY DEF CurrentTsLog

(3) UNCHANGED LogInNv BY DEF LogInNv
Extending sem pcr with Unhappy results in something other than SemHappy, which indicates that the log is not in the application pcr.

(3) \( \neg \text{LogInApp}' \)

(4) \( \text{semPcr}' \neq \text{SemHappy} \)

(5) USE DEF SemHappy

(5) USE DEF SemProtect

(5) USE DEF Pcri

(5) USE DEF Pcrx

(5) USE ThmPcrInitIsPcr

(5) \text{semPcr} = \text{SemProtect} BY DEF InvInSemProtect, InSem

(5) USE AssSemHappy

(5) QED BY ThmPcrExtendAnticollision

(4) QED BY DEF LogInApp

(3) UNCHANGED LogInTs

(4) USE DEF AllCurrentTs

(4) USE DEF CheckTsIsCurrent

(4) QED BY DEF LogInTs

(3) InvOneLog! goal! obtains' BY DEF IsOnLog, InvOneLog

(3) InvOneLog! goal! revokes' BY DEF IsOnLog, InvOneLog

(3) InvVerifiableRevocation' BY DEF InvVerifiableRevocation, InvOneLog

(3) QED BY DEF InvOneLog

NextSemChkpt2

(2) CASE NextSemChkpt2

(3) USE NextSemChkpt2

(3) USE DEF NextSemChkpt2

(3) UNCHANGED CheckTsIsCurrent(chkptts) BY DEF CheckTsIsCurrent

(3) UNCHANGED AllCurrentTs

(4) USE DEF AllCurrentTs

(4) USE DEF CheckTsIsCurrent

(4) QED BY DEF LogInTs

(3) UNCHANGED CurrentTsLog BY DEF CurrentTsLog

(3) UNCHANGED LogInNu

(4) USE DEF InvType

(4) USE DEF Nu

(4) QED BY DEF LogInNu

(3) UNCHANGED LogInApp BY DEF LogInApp

(3) UNCHANGED LogInTs BY DEF LogInTs

Storing the log from chkptts to the nu ram.

This depends on BugChkptSaveCurApp \( \triangleq \) FALSE

(3) \( \text{nv'.appPcr} = \text{chkptts.appPcr} \)

(4) BugChkptSaveCurApp = FALSE BY DEF BugChkptSaveCurApp

(4) QED BY DEF InvType, Nu

(3) InvOneLog! goal! obtains' BY DEF IsOnLog, InvOneLog
⟨3⟩ InvOneLog!goal!revokes' BY DEF IsOnLog, InvOneLog
⟨3⟩ InvVerifiableRevocation' BY DEF InvVerifiableRevocation, InvOneLog
⟨3⟩ QED BY DEF InvOneLog

NextSemChkpt3

(2) 19. CASE NextSemChkpt3
(3) USE NextSemChkpt3
(3) USE DEF NextSemChkpt3
(3) UNCHANGED LogInNv BY DEF LogInNv
(3) UNCHANGED LogInApp BY DEF LogInApp

Since no signed ts seal can have a bootCtr greater than the current bootCtr, incrementing bootCtr erases any log that might have been in a seal attestation.

This depends on BugChkptNoIncCtr \triangleq FALSE

⟨3⟩ ~LogInTs'
(4) USE DEF AllCurrentTs
(4) USE DEF CheckTsIsCurrent
(4) \forall ts \in tsvalues' \cup \{chkptts'\} : ts \in SignedTs \Rightarrow ts.bootCtr \neq bootCtr'
(5) TAKE ts \in tsvalues' \cup \{chkptts'\}
(5) ts \in tsvalues \cup \{chkptts\} OBVIOUS
(5) HAVE ts \in SignedTs
(5) ts.bootCtr \leq bootCtr' BY DEF InvSignedTsLeqBoot
(5) ts.bootCtr \in Nat BY DEF SignedTs
(5) bootCtr' \in Nat BY DEF InvType
(5) bootCtr < bootCtr' BY DEF InvType
(5) bootCtr < bootCtr' BY DEF BugChkptNoIncCtr = FALSE BY DEF BugChkptNoIncCtr
(6) QED BY ThmNatInc
(5) ts.bootCtr < bootCtr' BY ThmNatLeqLt
(5) QED BY ThmNatLeqXorGt, ThmNatLeqIsReflexive
(4) QED BY DEF LogInTs

Erasers all current ts seal attestations.

⟨3⟩ AllCurrentTs' = {} \wedge ~CheckTsIsCurrent(chkptts)'
(4) USE DEF AllCurrentTs
(4) USE DEF CheckTsIsCurrent
(4) QED BY DEF LogInTs

⟨3⟩ InvOneLog!goal!obtains' BY DEF IsOnLog, InvOneLog
⟨3⟩ InvOneLog!goal!revokes' BY DEF IsOnLog, InvOneLog
⟨3⟩ InvVerifiableRevocation' BY DEF InvVerifiableRevocation, InvOneLog
⟨3⟩ QED BY DEF InvOneLog

NextSemChkpt4
(2) 20. **CASE** NextSemChkpt4
(3) USE NextSemChkpt4
(3) USE DEF NextSemChkpt4
(3) UNCHANGED CheckTsIsCurrent(chkpts) BY DEF CheckTsIsCurrent
(3) UNCHANGED AllCurrentTs
  (4) USE DEF AllCurrentTs
  (4) USE DEF CheckTsIsCurrent
  (4) QED BY DEF LogInTs
(3) UNCHANGED CurrentTsLog BY DEF CurrentTsLog

Setting \( nv \) current indicates that the log is in the \( nv \) ram.

This depends on \( BugChkptNoSetCur \). \( \triangleq \) FALSE

(3) LogInNu'
  (4) \( nv'.current \)
    (5) \( BugChkptNoSetCur = \text{FALSE} \) BY DEF BugChkptNoSetCur
    (5) QED BY DEF InvType, Nu
  (4) QED BY DEF LogInNu
(3) UNCHANGED LogInApp BY DEF LogInApp
(3) UNCHANGED LogInTs BY DEF LogInTs

Since \( nv.current \) was changed, the prover needs to see the type of \( nv \) to know that the \( appPcr \) field did not change.

NextSemChkpt5

(2) 21. **CASE** NextSemChkpt5
(3) USE NextSemChkpt5
(3) USE DEF NextSemChkpt5
(3) UNCHANGED CheckTsIsCurrent(chkpts) BY DEF CheckTsIsCurrent
(3) UNCHANGED AllCurrentTs
  (4) USE DEF AllCurrentTs
  (4) USE DEF CheckTsIsCurrent
  (4) QED BY DEF LogInTs
(3) UNCHANGED CurrentTsLog BY DEF CurrentTsLog
(3) UNCHANGED LogInNu BY DEF LogInNu

Extending sem pcr with Unhappy results in something other than \( \text{SemHappy} \), which indicates that the log is not in the application pcr.

(3) \( \neg \text{LogInApp}' \)
  (4) \( \text{semPcr}' \neq \text{SemHappy} \)
    (5) USE DEF SemHappy
    (5) USE DEF SemProtect
    (5) USE DEF Pcri
    (5) USE DEF Pcrx
(5) USE ThmPcrInitIsPcr
(5) USE ThmPcrExtendIsPcr
(5) \( semPcr = SemProtect \) BY DEF InvInSemProtect, InSem
(5) USE AssSemHappy
(5) QED BY ThmPcrExtendAnticollision
(4) QED BY DEF LogInApp

\begin{align*}
\text{(1) QED BY DEF InvOneLog} \\
\text{(2) QED} \\
\text{BY (2)1,} \\
\text{(2)2, (2)3, (2)4, (2)5, (2)6, (2)8, (2)9,} \\
\text{(2)10, (2)11, (2)12, (2)13, (2)14, (2)15, (2)16, (2)17,} \\
\text{(2)18, (2)19, (2)20, (2)21} \\
\text{DEF Next} \\
\end{align*}

It is an invariant of the specification.

\textbf{THEOREM} ThmInvOneLog \triangleq \\
Spec \Rightarrow \square \text{InvOneLog}

\textbf{PROOF} \\
(1) \text{Init} \Rightarrow \text{InvOneLog} \text{BY ThmInitInvOneLog} \\
(1) \text{InvOneLog} \land [\text{Next}] \text{vars} \Rightarrow \text{InvOneLog}' \\
\text{BY ThmNextInvOneLog} \\
(1) \text{QED BY RuleINV1} \text{DEF Spec}
It is an invariant of the specification.  

**THEOREM** \( \text{ThmInvAccessUndeniability} \triangleq \)  
\( \text{Spec} \Rightarrow \Box \text{InvAccessUndeniability} \)  

**PROOF**  
(1) \( \text{InvOneLog} \Rightarrow \text{InvAccessUndeniability} \)  
(2) \( \text{HAVE InvOneLog} \)  
(2) \( \text{USE DEF InvOneLog} \)  
(2) \( \text{USE DEF InvAccessUndeniability} \)  
(2) \( \text{BugAuditNoCheckHappy} = \text{FALSE} \) \( \text{BY DEF BugAuditNoCheckHappy} \)  
(2) \( \text{CASE } \neg \text{LogInApp} \) \( \text{BY DEF LogInApp} \) \( \text{unable to audit} \)  
(2) \( \text{CASE } \text{LogInApp} \) \( \text{BY DEF IsOnLog} \) \( \text{audit} \)  
(2) \( \text{QED OBVIOUS} \)  

(1) \( \text{Spec} \Rightarrow \Box \text{InvOneLog} \) \( \text{BY ThmInvOneLog} \)  
(1) \( \text{QED} \)  

---  

**PROOF OF INVARIANT** \( \text{InvVerifiableRevocation} \)  

It is an invariant of the specification.  

**THEOREM** \( \text{ThmInvVerifiableRevocation} \triangleq \)  
\( \text{Spec} \Rightarrow \Box \text{InvVerifiableRevocation} \)  

**PROOF**  
(1) \( \text{InvOneLog} \Rightarrow \text{InvVerifiableRevocation} \)  
(2) \( \text{HAVE InvOneLog} \)  
(2) \( \text{USE DEF InvOneLog} \)  
(2) \( \text{QED OBVIOUS} \)  

(1) \( \text{Spec} \Rightarrow \Box \text{InvOneLog} \) \( \text{BY ThmInvOneLog} \)  
(1) \( \text{QED} \)