Cryptographically Sound Implementations for Typed Information-Flow Security

Cédric Fournet
Microsoft Research
MSR-INRIA Joint Centre
fournet@microsoft.com

Tamara Rezk
MSR–INRIA Joint Centre
Tamara.Rezk@inria.fr

Abstract

In language-based security, confidentiality and integrity policies conveniently specify the permitted flows of information between different parts of a program with diverse levels of trust. These policies enable a simple treatment of security, and they can often be verified by typing. However, their enforcement in concrete systems involves delicate compilation issues.

We consider cryptographic enforcement mechanisms for imperative programs with untrusted components. Such programs may represent, for instance, distributed systems connected by some untrusted network. In source programs, security depends on an abstract access-control policy for reading and writing the shared memory. In their implementations, shared memory is unprotected and security depends instead on encryption and signing.

We build a translation from well-typed source programs and policies to cryptographic implementations. To establish its correctness, we develop a type system for the target language. Our typing rules enforce a correct usage of cryptographic primitives against active adversaries; from an information-flow viewpoint, they capture controlled forms of robust declassification and endorsement. We show type soundness for a variant of the non-interference property, then show that our translation preserves typability.

We rely on concrete primitives and hypotheses for cryptography, stated in terms of probabilistic polynomial-time algorithms and games. We model these primitives as commands in our target language. Thus, we develop a uniform language-based model of security, ranging from computational non-interference for probabilistic programs down to standard cryptographic hypotheses.

Categories and Subject Descriptors D.2.0 [*Software Engineering*]: Protection Mechanisms; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Specification techniques.

General Terms Security, Verification, Design, Languages.

Keywords Secure information flow, confidentiality, integrity, noninterference, type systems, compilers, probabilistic programs, cryptography, computational model.

35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08), pp 323–335, January 7–12, 2008, San Francisco, CA, USA.

1. Introduction

One of the open challenges in security is to reliably protect program implementations by compilation (Abadi 1998). To this end, one needs languages that let the programmer specify security requirements and reason about them using simple abstractions, as well as tools that can produce code to enforce these requirements.

In particular, when considering the integrity and confidentiality of information, the verification of distributed programs entangles different aspects of system implementations, ranging from application-level information-flow control down to cryptographic algorithms and communication protocols (themselves depending on adequate integrity and confidentiality control for their cryptographic keys). Our thesis is that the cryptographic aspects should be left to the compiler.

In language-based information-flow security, confidentiality and integrity policies are specified using security labels, equipped with a partial order that describes permitted flows of information (Denning 1976; Zdancewic and Myers 2001). Security labels associated to program variables specify who can read from (confidentiality) and who can write to (integrity) a given variable. The preservation of confidentiality and integrity policies is expressed as *non-interference* properties, guaranteeing that the knowledge of an attacker with limited access to variables is not augmented by any program execution.

We consider cryptographic enforcement mechanisms for confidentiality and integrity in imperative programs. Our security model accounts for active adversaries, represented as untrusted (or unknown) parts of the program that may change unprotected memory during execution. The resulting programs may represent, for instance, distributed systems connected by some untrusted network, or untrusted machines containing protected subsystems. According to the program semantics, security depends on an abstract read/write policy for accessing shared memory. In their cryptographic implementation, shared memory is unprotected, and security depends instead on encryption and signing when accessing the shared memory.

A first basic example Consider two parties a and b that wish to perform some computation securely by exchanging a series of messages over some untrusted network. Using shared memory, we may write for instance

$$\_; (x := 1)^a; \_; (\mathsf{if}\ x\ \mathsf{then}\ y := 2\ \mathsf{else}\ y := z)^b; \_; (y := y+1)^a; \_$$

where the parentheses $(\,)^a$ and $(\,)^b$ indicate code that runs on behalf of a and b, respectively, and where the placeholders $\_$ stand for any untrusted code that may run in-between. Assuming that untrusted code does not access x, y, and z, we would expect for instance that z remains secret and $y = 3$ at the end of any run. To this end, a needs to securely pass x to b, then b needs to securely pass y to a.

In a less abstract setting, such flows of information between variables may involve communication over untrusted channels, e.g. shared memory that may be read and modified by active adversaries, with some adequate encryption and signing. For example, assuming the variables x_e, x_s and y_e, y_s are used to pass the encrypted values and signatures for x and y, respectively, the second command $(\mathsf{if}\ x\ \mathsf{then}\ y := 2\ \mathsf{else}\ y := z)^b$ may be implemented as

$$\begin{array}{l}\mathsf{if}\ \mathcal{V}(x_e, x_s, k_v)\ \mathsf{then}\ (\\ \quad x^b := \mathcal{D}(x_e, k_d);\\ \quad \mathsf{if}\ x^b\ \mathsf{then}\ y^b := 2\ \mathsf{else}\ y^b := z^b;\\ \quad y_e := \mathcal{E}(y^b, k_e); y_s := \mathcal{S}(y_e, k_s))\end{array}$$

where, in order to read x out of its wire format x_e, x_s, the code first verifies ($\mathcal{V}$) the signature then performs a decryption ($\mathcal{D}$) to extract a local copy of x into x^b; and, conversely, for writing y, the code first encrypts ($\mathcal{E}$) its updated copy y^b and then signs ($\mathcal{S}$) the encrypted value.

This implementation code does not rely on the confidentiality or integrity of the shared variables used on the wire (variables x_e, x_s, y_e, y_s). Instead, it relies on the adequate generation and management of the keys used for verifying, decrypting, encrypting, and signing, as well as security assumptions on the cryptographic primitives. If we rely on a public-key signature scheme, for instance, the integrity of the verification key k_v must be higher than the integrity of x, while the confidentiality of the signing key k_s must be high enough to protect the integrity of y. Also, if the keys are used for other purposes (and we can hardly dedicate 4 keys to every variable), we need to carefully control their interaction. For instance, in the code above, we cannot use the same keys for protecting x and y, as an adversary may then achieve $y = 2$ at the end of the computation by inserting the code $y_e := x_e; y_s := x_s$ between b and a. Besides, we cannot assume that the computation always completes successfully, as indeed an adversary may insert $y_s := 0$ before b's code and thus cause the signature verification to fail, so we also need to qualify our notion of integrity for this computation.

Symbolic versus computational cryptography In contrast with most language-based approaches, we do not rely on symbolic "black box" cryptography. Despite considerable successes for protocol verification, symbolic cryptography may be dangerously abstract for protocol design, especially as regards indistinguishability properties and information-flow. For example, cryptographic algorithms do not actually guarantee the confidentiality of their keys as values—only that not enough information is leaked to effectively recover an encrypted payload or fake a signature (see e.g. Abadi and Rogaway 2002). Hence, under standard assumptions, an adversary may learn which keys are used by traffic analysis, potentially opening a side channel if key selection depends on a secret guard. Instead of symbolic cryptography, we use a concrete model with probabilistic polynomially-bounded algorithms on bitstrings, and standard security hypotheses (IND-CCA2, IND-CMA). Thus, we gain a more precise and realistic information-flow result: we guarantee that the probability that any given polynomial-time adversary illegally obtains (or influences) information becomes negligible as the keys get long enough.

Our contributions

1. Starting from an imperative language with information-flow policies for both confidentiality and integrity, we adapt a simple type system for non-interference to accommodate a "fail-stop" semantics for runtime checks in the presence of active attackers.
2. We develop a target language for implementations that rely on cryptography, with a probabilistic semantics. We use it to conveniently express cryptographic algorithms, active adversaries, and oracles as well as our implementation code in a precise setting. We can thus recast standard cryptographic assumptions as properties on probabilistic programs. In order to reconcile this style of properties with classic information-flow properties in a uniform framework, we also reformulate non-interference more syntactically, as a game coded in our language with explicit commands for programs, active adversaries, and tests.
3. We equip this probabilistic language with a type system for checking the usage of cryptography. From an information viewpoint, we capture controlled forms of declassification (downgrading of confidentiality levels of variables) after encryption and endorsement (upgrading of integrity levels of variables) after signature verification. We regard our type system as a tool for structuring cryptographic proofs. Indeed, the game-rewriting arguments in the main proofs are expressed as typed program transformations. To our knowledge, this is the first computationally sound type system for cryptographic information flow that can handle active adversaries.
4. We give a typed translation from the simple language to the target language. We show that, if a source program is typable, then its translation is also typable, hence it has the property of computational non-interference against probabilistic polynomial-time active adversaries. To our knowledge, this is the first cryptographic translation for general information-flow security.

Limitations Our results apply to a large class of protocols and program translations, but they still have important limitations from a general programming viewpoint. For example, we do not model concurrency to avoid the complications of non-determinism in computational models (see e.g. Adão and Fournet 2006). Also, as in prior types for computational cryptography, we separately keep track of every key, which excludes key generations within loops.

Our theorems rely on global conditions: that programs are polynomial, and that some variables are initialized before being read, and assigned only in specific parts of the code. We do not enforce these conditions by typing; they can be checked independently, and sometimes achieved by preliminary program transformations.

Related work Laud (2001) pioneers work on information flow relying on concrete cryptographic assumptions. He introduces computational non-interference for encryption in a model with passive adversaries. Our definitions generalize this property to the active case, with adversaries that may interfere with the normal execution of programs, and also covers integrity properties. Backes (2005) relates negligible information flows to computational non-interference and shows that this property is preserved under simulatability in reactive systems.

Laud and Vene (2005) propose a type system to verify computational non-interference against passive adversaries in an imperative language with symmetric encryption and dynamic key generation. Their types are more precise than ours for tracking key dependencies, but their system does not enable to type decrypted values as keys (cf. Example 8). In a similar line of work, Smith and Alpízar (2006) present a type system with encryption and decryption but no explicit keys; they assume a single implicit key-pair for the whole program, which leads to a clean type-system presentation.

Using symbolic cryptography, Askarov et al. (2006) generalize non-interference to allow flows that arise from encryption. In their model, a type system enforces secure cryptographically masked flows for a non-deterministic semantics of encryption. Recently, Laud (2008) investigates conditions such that cryptographically masked flows imply security in the computational model.

Integrity and active adversaries Several works consider the interaction between confidentiality and integrity policies in the presence of active adversaries. (This interaction plays a central role in our

handling of cryptographic keys.) Zdancewic and Myers (2001) and Myers et al. (2006) propose general definitions for non-interference with active adversaries. Their definitions are close to ours, but do not consider cryptography.

Secure implementations for information flow Jif/Split (Zdancewic et al. 2002; Zheng et al. 2003) is a compiler that implements secure distributed systems on mutually untrusted hosts from sequential programs annotated with information flow types. The implementation assumes that all communications are private. Our work can be seen as an attempt to implement (and verify) a cryptographic back-end for Jif/Split. In principle, many of their techniques could be applied (for instance, as type-preserving transformations) before applying our translation, to deal with other aspects of distribution such as global control flow.

Vaughan and Zdancewic (2007) design a typed language with high-level dynamic "pack" and "unpack" security primitives, and describe their implementation using authenticated encryption. Their language also uses labels that combine confidentiality and integrity, expressed in the decentralized label model (Myers and Liskov 2000). The correctness of their implementation is proved for a symbolic model of cryptography with passive attackers (who can observe encrypted memory after execution of the program).

Cryptographic implementations for programming languages We mention related work only within the computational model of cryptography. Backes et al. (2003) provide a sound execution framework for protocols that use an idealized cryptographic library. Laud (2005) designs a typed cryptographic language and implements it on top of their framework. His type system also keeps track of key confidentiality and integrity, in order to meet the hypotheses of Backes et al. (2003) and guarantee *payload secrecy*, which can be interpreted as a form of computational non-interference. Abadi et al. (2006) rely on Laud's language to compile security-typed variants of the pi calculus and thereby obtain payload secrecy (but no integrity) for their distributed implementations.

Adão and Fournet (2006) design a process calculus with abstractions for secure communications (but no explicit cryptography) and establish the computational soundness of its implementation for trace and equivalence properties.

Contents Section 2 defines our source language, policies, security properties, and type systems. Section 3 defines our target language and security properties—computational non-interference—and explains our cryptographic assumptions. Section 4 presents our cryptographic type system. Section 5 presents our type-safe translation. Section 6 concludes and discusses future work.

Additional details appear in a companion paper at `http://www.msr-inria.inria.fr/projects/sec/cflow`.

2. Security policies and non-interference

We present a simple imperative while language and equip it with security policies. We recall standard notions of non-interference for confidentiality and integrity, and a simple type system for checking this property. In preparation for our cryptographic implementation, we then extend these notions to active adversaries.

2.1 A simple imperative while language

Our source programs consist of expressions and commands, with the following grammar:

$$e \ ::= \ x \mid v \mid op(e_1, \ldots, e_n)$$
$$P \ ::= \ x := e \mid P; P \mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P \mid \mathsf{while}\ e\ \mathsf{do}\ P \mid \mathsf{skip}$$

where x ranges over variables, v ranges over literal values (for now bitstrings, which may represent integers, strings, booleans), and op ranges over n-ary functions (such as arithmetic, string, and boolean operations). For completeness, we assume that these functions include all standard bitwise operations on values.

We write $\mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2; P_3$ for $(\mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2); P_3$ and write $\mathsf{if}\ e\ \mathsf{then}\ P$ for $\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else\ skip}$. We let $rv(P)$ and $wv(P)$ be the sets of variables that are syntactically read and written by P: $x \in rv(P)$ when x occurs in an expression of P; and $x \in wv(P)$ when a command of the form $x := e$ occurs in P. We let $v(P) = rv(P) \cup wv(P)$.

We let μ range over memories, that is, finite functions from variables to values plus a special term $\perp$. We denote by $\mu(x) = \perp$ that variable x is not initialized in memory μ. We assume that $\perp$ does not occur in commands. We write $\mu\{x \mapsto v\}$ for the memory that maps x to v and any $y \neq x$ to $\mu(y)$. For a given memory domain, we let $\mu_\perp$ be the memory that maps every variable to $\perp$.

We use a standard semantics for programs (formally defined as a special case in Section 3). Configurations range over pairs of a command and a memory, written $\langle P, \mu \rangle$, plus inert configurations, written $\langle \surd, \mu \rangle$ or just μ, that represent command termination with final memory μ. We implicitly assume that all the variables of P are in the domain of μ. We let $\rightsquigarrow$ represent single-step execution of commands between configurations and let $\rightsquigarrow^*$ be its reflexive transitive closure. We denote normal termination as $\langle P, \mu \rangle \Downarrow \mu'$, that is, starting with initial memory μ, command P completes with final memory μ' in any number of steps.

2.2 Non-interference against passive adversaries (review)

We annotate variables, expressions, and commands with security labels. These labels specify the programmer's security intent, but they do not affect the operational semantics.

The labels form a lattice $(\mathcal{L}, \leq)$, obtained as the product of two lattices of confidentiality levels $(\mathcal{L}_C, \leq_C)$ and integrity levels $(\mathcal{L}_I, \leq_I)$. We write $\perp_\mathcal{L}$ and $\top_\mathcal{L}$ for the smallest and largest elements of $\mathcal{L}$, and $\sqcup$ for the least upper bound of two elements of $\mathcal{L}$. For a given label $\ell = (\ell_C, \ell_I)$ of $\mathcal{L}$, the confidentiality level ℓ_C specifies a read level for variables, while the integrity level ℓ_I specifies a write level; the meaning of $\ell \leq \ell'$ is that ℓ' is more confidential (can be read by fewer commands) and less integral (can be written by more commands) than ℓ. We let $C(\ell) = \ell_C$ and $I(\ell) = \ell_I$ be the projections that yield the confidentiality and integrity parts of a label. Hence, the partial order on $\mathcal{L}$ is defined as $\ell \leq \ell'$ iff $C(\ell) \leq_C C(\ell')$ and $I(\ell) \leq_I I(\ell')$. We refer to the decentralized label model of Myers and Liskov (2000) and Vaughan and Zdancewic (2007) for a concrete syntax for setting such lattices.

We represent our security policies as functions Γ from variables to security types τ of the form $t(\ell)$, where t (for now) can only be instantiated with data type Data and where ℓ is a security label. Overloading our notations, we lift our confidentiality and integrity projections C and I from labels to security types, and let $C(t(\ell)) = C(\ell)$ (resp. $I(t(\ell)) = I(\ell)$). We also let $T(t(\ell)) = t$. We now proceed to define non-interference for confidentiality and integrity, relative to a given policy.

DEFINITION 1 (Memory indistinguishability). *Let V be a set of variables. The memories μ_0 and μ_1 are indistinguishable on V, written $\mu_0 \sim_V \mu_1$, when $x \in V$ implies $\mu_0(x) = \mu_1(x)$.*

DEFINITION 2 (Non-interference on V). *The command P is non-interferent on V when, for all memories μ_0 and μ_1, if $\mu_0 \sim_V \mu_1$ and $\langle P, \mu_b \rangle \Downarrow \mu'_b$ for $b = 0, 1$, then $\mu'_0 \sim_V \mu'_1$.*

Intuitively, as long as it terminates, a non-interferent command does not leak any information from the hidden part of its initial memory (outside V) to the visible part of its final memory (inside V). (In this work, we consider only the termination-insensitive variant of non-interference. Indeed, in our refined cryptographic

model, we are going to demand that all commands always terminate in polynomial time, thereby excluding any termination leak.)

Standard definitions of information-flow security against passive adversaries are obtained from Definition 2 by letting the set of observed variables V collect either the low-confidentiality variables or the high-integrity variables:

DEFINITION 3 (Non-interference at α, passive case). *Let Γ be a memory policy and $\alpha \in \mathcal{L}$ a security label. Let*

$$V_\alpha^C = \{x \mid C(\Gamma(x)) \leq_C C(\alpha)\} \quad V_\alpha^I = \{x \mid I(\Gamma(x)) \leq_I I(\alpha)\}$$

The command P preserves confidentiality at α when it is non-interferent on V_α^C; it preserves integrity at α when it is non-interferent on V_α^I.

2.3 Active adversaries and runtime errors

In terms of attacker model, memory indistinguishability accounts for adversaries that may partially observe the outcome of the computation but do not interfere with it. More generally, we are interested in non-interference for programs that may include commands representing active adversaries. Our approach largely follows *robust declassification* (Myers et al. 2006).

For a given $\alpha \in \mathcal{L}$, an α-adversary is a command, say A, that reads variables with confidentiality level less than or equal to $C(\alpha)$ and writes variables with integrity level not less than or equal to $I(\alpha)$:

$$rv(A) \subseteq V_\alpha^C \qquad wv(A) \cap V_\alpha^I = \emptyset$$

In the rest of the paper, we let α denote the security level of the adversary. As usual, once α is fixed, we may restrict our attention to the product of binary lattices $L \leq_C H$ for confidentiality and $H \leq_I L$ for integrity, with just four labels *HL*, *HH*, *LL*, and *LH*, and set $\alpha = LH$. We use this 4-point lattice in examples. On the other hand, we intend to develop implementations that do not a priori depend on a fixed α.

We consider programs obtained by composing commands with diverse levels of trust, including e.g. arbitrary α-adversaries as well as fixed, trusted commands. To this end, we write $P[\_]$ for a command context (with a grammar obtained from that of P by adding a hole $\_$) and $P[P']$ for the command obtained by replacing each occurrence of $\_$ with P'. We also use n-ary command contexts, with n distinct holes, and write $P[\vec{P'}]$ for the command obtained by instantiating these holes with the vector of commands $\vec{P'}$.

Although our language does not feature procedure calls, we can use command contexts to model arbitrary commands with access to fixed, privileged procedures, sometimes called "oracles" in cryptography, using fixed variables for passing their input and output parameters. For example, the command $P_0; [A[P']]$ represents a command that first runs initialization code P_0 then runs A, which in turn may invoke command P' any number of times.

Handling runtime errors The integrity of a run may clearly be affected by an active adversary that can write into low-integrity variables. Consider for instance the command context

$$P[\_, Q] \doteq l := 4; \_; \mathsf{if}\ l = 4\ \mathsf{then}\ h := 10\ \mathsf{else}\ Q$$

where the hole $\_$ stands for low-level code. After running command $P[\mathsf{skip}, h := 5]$ we have $h = 10$, but this is not the case with command $P[l := 0, h := 5]$, as there is an implicit flow from l to h. Hence, if h has high integrity and l has not, the command $P[l := 0, h := 5]$, and even $P[\mathsf{skip}, \mathsf{skip}]$, are typically rejected by type systems for non-interference.

This approach is too restrictive in our case, as we expect the adversary to be able to modify signed or encrypted values, as long as our cryptographic implementation can catch the attack as a runtime error—typically, any signature verification is a low-integrity guard. Accordingly, we relax our security definitions to accept command contexts such as $P[\_, \mathsf{skip}]$, even if l is less integral than h, on the following ground: if h is ever assigned, its value will be 10, so its integrity is preserved.

In the following, we interpret any read of an uninitialized variable as a runtime error. In the example above, a run of command $P[l := 0, \mathsf{skip}]$ (starting with memory $l \mapsto \bot, h \mapsto \bot$) leaves h uninitialized. Further, we assume that programs never read variables that are uninitialized and not writable by the adversary ($I(x) \leq_I I(\alpha)$). This property can be enforced independently, for instance by relying on static analyses (as discussed by Laud and Vene 2005).

We relax our notion of indistinguishability (Definition 1) to disregard the observation of uninitialized variables:

DEFINITION 4 (Weak memory indistinguishability). *Let V be a set of variables. The memories μ_0 and μ_1 are weakly indistinguishable on V, written $\mu_0 \sim_V^\perp \mu_1$, when $x \in V$ implies either $\mu_0(x) = \mu_1(x)$, or $\mu_0(x) = \bot$, or $\mu_1(x) = \bot$.*

We also adapt non-interference (Definition 2) to account for weak indistinguishability. To this end, we further distinguish a set U of variables that must be left uninitialized in initial memories. We intend this set to gather high-integrity variables exclusively written by P.

DEFINITION 5 (Weak non-interference on V, U). *Let V and U be two sets of variables. The command P is weakly non-interferent on V, U when, for all memories μ_0 and μ_1, if*

1. *$\mu_0(x) = \mu_1(x) = \bot$ for every $x \in U$,*
2. *$\mu_0 \sim_V^\perp \mu_1$, and*
3. *$\langle P, \mu_b \rangle \Downarrow \mu_b'$ for $b = 0, 1$,*

then $\mu_0' \sim_V^\perp \mu_1'$.

DEFINITION 6 (Weak non-interference at α). *Let Γ be a memory policy and $\alpha \in \mathcal{L}$ a security label. The command P is weakly non-interferent at α when P is weakly non-interferent on both $V_\alpha^C, \emptyset$ and $V_\alpha^I, V_\alpha^I \cap wv(P)$.*

Intuitively, the security property for active adversaries reflects that, even if an adversary may prevent the normal completion of a command (leaving more uninitialized variables in the final memories), he will neither learn more than by eavesdropping complete runs of the command, nor be able to affect the final value of defined high-integrity variables. In preparation for cryptographic refinements, we express this security property for command contexts P with holes for active adversaries in a style close to the one used for cryptographic games (see e.g. Bellare and Rogaway 2004) with subcommands that explicitly code initial memories, adversaries, and observations.

DEFINITION 7 (Non-interference against active adversaries). *Let Γ be a memory policy and $\alpha \in \mathcal{L}$ a security label.*

The command context P is non-interferent against α-adversaries when, for both $V, U = V_\alpha^C, \emptyset$ and $V, U = V_\alpha^I, V_\alpha^I \cap wv(P)$, and for all commands

- *J writing $V \setminus U$: $wv(J) \subseteq V \setminus U$;*
- *B_b for $b = 0, 1$ writing outside V: $wv(B_b) \cap V = \emptyset$;*
- *$\vec{A}$ α-adversaries;*
- *T reading V, writing g: $rv(T) \subseteq V$; $g \notin wv(J, B_0, B_1, \vec{A})$;*

the value of g after running $J; B_b; P[\vec{A}]; T$ does not depend on b: if $\langle J; B_b; P[\vec{A}], \mu_\perp \rangle \Downarrow \mu_b'$, $\bigwedge_{x \in rv(T)} \mu_b'(x) \neq \bot$, and $\langle T, \mu_b' \rangle \Downarrow \mu_b''$ for $b = 0, 1$ then we have $\mu_0''(g) = \mu_1''(g)$.

The command $G_b = J; B_b; P[\vec{A}]; T$ represents a game, parameterized by a meta-variable b that is either 0 or 1; its first part $J; B_b$ initializes variables and, depending on b, yields two indistinguishable memories (μ_0 and μ_1 in Definition 2); the second part $P[\vec{A}]$ runs the command context P in combination with active adversaries $\vec{A}$; this yields two memories (μ'_0 and μ'_1 in Definition 2); the final part of the command, T, represents an observer that attempts to guess the value of b from memory μ'_b. Intuitively, an opponent player that chooses the commands J, B_0, B_1, $\vec{A}$, and T wins when $g = 0$ after running G_0 and $g = 1$ after running G_1.

EXAMPLE 1 (Non-interference for a simple lattice). *Let $\mathcal{L}$ be the 4-point lattice with labels HL, HH, LL, and LH. Let $\alpha = LH$.*

Let P be a command context that writes $wv(P) = \{x_{HH}, x_{HL}, x_{LL}, x'_{LH}\}$ and reads $rv(P) = wv(P) \cup \{x_{LH}\}$, with a single hole. To show that P is non-interferent against α-adversaries, we check that g does not depend on the choice between B_0 and B_1 for any commands that operate on the variables of P as follows:

- *for the confidentiality game:*
  - *J sets x_{LH}, x_{LL}, and x'_{LH};*
  - *B_b set x_{HL} and x_{HH} depending on x_{LH}, x_{LL}, and x'_{LH};*
  - *A sets x_{HL} and x_{LL} depending on x_{LL}, x_{LH}, and x'_{LH};*
  - *T sets g depending on x_{LH}, x'_{LH}, and x_{LL};*
- *for the integrity game:*
  - *J sets x_{LH};*
  - *B_b set x_{HL} and x_{LL} depending on x_{LH};*
  - *A sets x_{HL} and x_{LL} depending on x_{LL}, x_{LH}, and x'_{LH};*
  - *T sets g depending on x_{LH} and, only when they are initialized, on x'_{LH} and x_{HH}.*

EXAMPLE 2. *According to Definition 7, the command context $y_{LH} := x_{HH}; y_{LH} := 0$ is secure, but $y_{LH} := x_{HH}; \_; y_{LH} := 0$ (with an intermediate adversary) is not secure, since for instance*

$$J \doteq \mathsf{skip} \qquad A \doteq z_{LL} := y_{LH}$$
$$B_b \doteq x_{HH} := b \qquad T \doteq g := z_{LL}$$

break the confidentiality game, leaving $g = b$.

The following lemma relates non-interference for passive and active adversaries, in case the command P has no holes:

LEMMA 1. *A command P is weakly non-interferent at α if and only if P (as a command context P with no _) is non-interferent against active adversaries.*

In the general case, when P has at least one hole, non-interference against active adversaries implies that $P[\vec{\mathsf{skip}}]$ is weakly non-interferent, but the converse may not hold, as can be seen on Examples 2 (due to an explicit confidentiality flow) and 3 (due to an implicit integrity flow):

EXAMPLE 3. *For the lattice of Example 1, consider the context*

$$P = x_{LL} := 0; \_; \mathsf{if}\ x_{LL} = 4\ \mathsf{then}\ y_{LH} := 1\ \mathsf{else}\ y_{LH} := 0$$

$P[\mathsf{skip}]$ is weakly non-interferent, but $P[\_]$ is interferent against active adversaries for the integrity game, since we can define

$$J \doteq \mathsf{skip} \qquad A \doteq \mathsf{if}\ z_{LL} = 1\ \mathsf{then}\ x_{LL} := 4$$
$$B_b \doteq z_{LL} := b \qquad T \doteq g := y_{LH}$$

2.4 A simple type system for non-interference

Type systems for controlling information flows have been widely studied in the literature (see Sabelfeld and Myers 2003 for a survey). We recall a standard type system for establishing non-interference (Definition 2) and then adapt it for establishing non-interference against active adversaries (Definition 7). The resulting type system is further extended in Section 4 to account for cryptographic primitives.

$$\textsc{Var}\ \frac{}{\vdash x : \Gamma(x)} \qquad \textsc{Val}\ \frac{}{\vdash v : \mathsf{Data}\,(\bot^{\mathcal{L}})}$$

$$\textsc{Op}\ \frac{\vdash e_i : \mathsf{Data}\,(\ell)\ \text{ for } i = 1..n}{\vdash op(e_1, \ldots, e_n) : \mathsf{Data}\,(\ell)} \qquad \textsc{SubE}\ \frac{\vdash e : \tau \qquad \tau \leq \tau'}{\vdash e : \tau'}$$

Figure 1. Typing rules for source expressions with policy Γ.

$$\textsc{Assign}\ \frac{\vdash e : \Gamma(x)}{\vdash x := e : L(x)} \qquad \textsc{Seq}\ \frac{\vdash P : \ell \qquad \vdash P' : \ell}{\vdash P; P' : \ell}$$

$$\textsc{Cond}\ \frac{\vdash e : \mathsf{Data}\,(\ell) \qquad \vdash P : \ell \qquad \vdash P' : \ell}{\vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P' : \ell} \qquad \textsc{Skip}\ \vdash \mathsf{skip} : \top^{\mathcal{L}}$$

$$\textsc{While}\ \frac{\vdash e : \mathsf{Data}\,(\ell) \qquad \vdash P : \ell}{\vdash \mathsf{while}\ e\ \mathsf{do}\ P : \ell} \qquad \textsc{SubC}\ \frac{\vdash P : \ell \qquad \ell' \leq \ell}{\vdash P : \ell'}$$

Figure 2. Typing rules for source commands with policy Γ.

$$\textsc{Check}\ \frac{\vdash e : \mathsf{Data}\,(\ell') \qquad C(\ell') \leq_C C(\ell) \qquad \vdash P : \ell}{\vdash \mathsf{if}\ e\ \mathsf{then}\ P : \ell} \qquad \textsc{Hole}\ \vdash \_ : \top^{\mathcal{L}}$$

Figure 3. Additional typing rules for source command contexts.

Recall that security types are of the form $t(\ell)$, where t is just Data for now and $\ell \in \mathcal{L}$ is a security label. We use L as the projection from types to security labels, that is $L(t(\ell)) = \ell$. For brevity, in the typing rules we sometimes abbreviate type parameters $\Gamma(x)$ to x. We lift the security preorder $\leq$ from labels to types, and let $t(\ell) \leq t'(\ell')$ iff $t = t'$ and $\ell \leq \ell'$.

For a given security policy Γ, typing judgments for expressions, written $\Gamma \vdash e : t(\ell)$, mean that e reads variables of level at most ℓ. Typing judgments for commands, written $\Gamma \vdash P : \ell$, mean that P is secure and writes variables of level at least ℓ. In the following, we often omit the fixed policy Γ in typing judgments. The typing rules appear in Figures 1 and 2, respectively. We also write $\Gamma \vdash P$ to denote typability of P with policy Γ, that is, there exists ℓ such that $\vdash P : \ell$.

The theorem below states that this simple type system is sound with regard to non-interference (Definition 2); its proof is a simple induction on the number of reduction steps.

THEOREM 2. *Let Γ be a security policy and $\alpha \in \mathcal{L}$ a security label. If $\Gamma \vdash P$, then P is non-interferent at α.*

We extend our type system to account for active adversaries. We type command contexts with placeholders that stand for α-adversaries by supplementing the rules of Figure 2 with those given in Figure 3. (The rules implicitly apply for some fixed policy Γ.)

Rule HOLE types placeholders for α-adversaries with $\top^{\mathcal{L}}$. Although the rule does not enforce any restriction, Definition 7 allows only α-adversaries to be placed in holes.

Rule CHECK allows some implicit integrity flows from conditional expressions to the then branch P when there is no else branch. The rule usefully applies to error handling, as discussed in Section 2.3, when the adversary controls the conditional execution of P; it is sound only together with an additional property, given

below, that restricts the high-integrity variables that P may write. This global, syntactic property demands that any variable $x \in V$ be written by at most one of a series of subcommands, thereby guaranteeing that $x = \perp$ if this subcommand is not executed.

DEFINITION 8 (Exclusive assignments). *Let V be a set of variables. Let $P_1, \ldots, P_n$ be subcommands of P, that is, $P = Q_i[P_i]$ for some command context Q_i for $i = 1..n$.*

$P_1, \ldots, P_n$ exclusively assign V in P when the hole of Q_i is not within any while loop and $V \cap wv(P_i) \cap wv(Q_i) = \emptyset$ for $i = 1..n$.

The next theorem states the soundness of the extended type system for command contexts with holes representing adversaries.

THEOREM 3. *Let Γ be a policy and $\alpha \in \mathcal{L}$ a security label. Assume $\Gamma \vdash P$ and all commands P' that occur in commands* if e then P' *typed by* CHECK *exclusively assign V_α^I in P.*

Then P is non-interferent against active α-adversaries.

3. Target cryptographic language and computational non-interference

Next, we add probabilistic primitives, define our target security property as a refinement of Definition 7, and specify cryptographic primitives and hypotheses.

3.1 A probabilistic language

The target language extends our imperative language with probabilistic functions, ranged over by f.

$$P \quad ::= \quad \ldots \mid x_1, \ldots, x_m := f(y_1, \ldots, y_n)$$

For simplicity, these probabilistic functions may occur only at top level in commands (so that expressions remain deterministic). We let $rv(\vec{x} := f(\vec{y})) = \{\vec{y}\}$ and $wv(\vec{x} := f(\vec{y})) = \{\vec{x}\}$.

Every function f is equipped with an associated parametric probability distribution $[\![f]\!]$. (Hence, in the special case f is a deterministic function, the distribution $[\![f]\!](\vec{v})$ gives probability 1 to $f(\vec{v})$ and 0 to any other output.) We write $\{0,1\}$ for the "coin-tossing" function that returns either 0 or 1 with probability $\frac{1}{2}$.

Instead of single configurations $\langle P, \mu \rangle$, we now consider distributions of configurations, ranged over by d. The operational semantics for commands is given in Figure 4, as a Markov chain (Hermanns 2002) on the set of all configurations, written S, with probabilistic steps induced by the function distributions. (Formally, rule STABLE guarantees that all states have leaving transitions whose probabilities sum to 1.) We omit the usual semantics for expressions $[\![e]\!](\mu)$.

An initial distribution for P is a distribution of configurations that has all its weight on configurations of the form $\langle P, \mu \rangle$. Hence, we can lift initial memories to distributions and define the transition system as a transition system from input distributions to output distributions (see e.g. Monniaux 2001).

DEFINITION 9. *Let* $\mathsf{prob} : S \times S \mapsto [0,1]$ *such that $s \rightsquigarrow_{\mathsf{prob}(s,s')} s'$ in Figure 4 and $\mathcal{T}$ be the distribution transformer such that*

$$\mathcal{T}(d)(s') = \textstyle\sum_{s \in S} \mathsf{prob}(s, s') d(s)$$

The semantics of a probabilistic program is given by a sequence of distribution transformations, starting from an initial distribution d_0. We let $d_i \rightsquigarrow d_{i+1}$ when $d_{i+1} = \mathcal{T}(d_i)$ and let $\rightsquigarrow^*$ be the transitive clausure of $\rightsquigarrow$. The probability that a program P terminates after n steps starting with initial distribution d_0 is $p_n = \sum_{s=\langle \surd, \mu \rangle} d_n(s)$.

We let $\Pr[P; \varphi]$ be the probability that, starting from a given initial distribution d_0, command P completes with a final memory that meets condition φ, that is, $\Pr[P; \varphi] = \lim_{n \geq 0} \sum_{s=\langle \surd, \mu \rangle \mid \varphi} d_n(s)$. (The limit exists as the sum increases with n and is bounded by 1.)

3.2 Polynomial-time assumptions

Our commands capture exactly the algorithms that can be coded on probabilistic Turing machines, using shared memory as input and output tapes. Further, polynomial runs of commands correspond to polynomial runs of these machines. Thus, we can recast standard cryptographic assumptions and games in the formal setting of this language, quantifying for instance over all polynomial-time commands to represent all polynomial adversaries.

In the following definitions, we assume that the initial distribution for a given command P is $d_0^\perp$, which gives all its weight to the uninitialized state $\langle P, \mu_\perp \rangle$.

We assume that all algorithms are probabilistic and computable in time bounded by some polynomials in η, the security parameter. Intuitively, η represents the lengths of the keys. In order to avoid passing η explicitly, we assume that x_η is a read-only variable initialized with the security parameter.

We assume that all primitive operations are polynomial in their parameters and that the distribution for all our probabilistic primitive functions are polynomial-time samplable (so that any polynomial program that calls these primitives could also be written as a polynomial program that includes its own implementation of these primitives as subcommands).

In the rest of the paper, rather than distributions, we consider families of distributions parameterized by η (written $d(\eta)$), also known as ensembles. Thus, for a fixed domain of variables, the initial distribution $d_0^\perp$ becomes the family of distributions $d_0^\perp(\eta)$ where all the weight is given to $\langle P, \mu_\perp\{x_\eta \mapsto \eta\} \rangle$. We overload $\Pr[P; \varphi]$ to denote the probability function parameterized by η.

Security properties are often expressed in terms of games, coded as commands that sample a secret boolean $b := \{0,1\}$ then interact with adversary commands. The goal of the adversary commands is to write into some variable g its guess as to the value of b: the adversary wins when $b = g$. The trivial adversary $g := \{0,1\}$ wins with probability $\frac{1}{2}$, so we are interested in the *advantage* of an adversary, defined as the probability that $b = g$ minus $\frac{1}{2}$.

For any given η, the adversary may guess any secret variables with a non-zero probability, including variables that store cryptographic keys. Thus, in contrast with our definitions of non-interference so far, we cannot expect the advantage to be 0 as we start relying on cryptography. Rather, we expect this advantage to be a negligible function of η. We recall the definition of negligible functions:

DEFINITION 10 (Negligible function). *A function $f : \mathbb{N} \to \mathbb{R}$ is* negligible *when, for all $c > 0$ there exists n_c such that, for all $n \geq n_c$, we have $f(n) \leq n^{-c}$.*

3.3 Computational non-interference

We refine our notions of non-interference to account for probabilities, and in particular for the possibility that some information is leaked (or corrupted) with a negligible probability. Instead of running two commands for $b = 0, 1$, we run a single probabilistic command that first picks b uniformally at random. We begin with a probabilistic, code-based variant of non-interference against passive adversaries (Definition 2) similar to the one introduced by Laud (2001).

DEFINITION 11 (Computational non-interference on V, U). *The polynomial command P is computationally non-interferent on V, U when for all polynomial commands*

- *J writing $V \setminus U$: $wv(J) \subseteq V \setminus U$;*
- *B_b for $b = 0, 1$ writing outside V: $wv(B_b) \cap (V \cup U) = \emptyset$;*
- *T reading V, writing g: $rv(T) \subseteq V$; $g \notin wv(J, B_0, B_1, \vec{A})$;*

$$\textsc{AssignS}\ \frac{[\![e]\!](\mu) = v}{\langle x := e, \mu\rangle \rightsquigarrow_1 \langle \surd, \mu\{x \mapsto v\}\rangle} \qquad \textsc{SeqS}\ \frac{\langle P, \mu\rangle \rightsquigarrow_p \langle P_1, \mu_1\rangle \qquad P_1 \neq \surd}{\langle P; P', \mu\rangle \rightsquigarrow_p \langle P_1; P', \mu_1\rangle} \qquad \textsc{SeqT}\ \frac{\langle P, \mu\rangle \rightsquigarrow_p \langle \surd, \mu_1\rangle}{\langle P; P', \mu\rangle \rightsquigarrow_p \langle P', \mu_1\rangle} \qquad \textsc{SkipS}\ \langle \mathsf{skip}, \mu\rangle \rightsquigarrow_1 \langle \surd, \mu\rangle$$

$$\textsc{Stable}\ \langle \surd, \mu\rangle \rightsquigarrow_1 \langle \surd, \mu\rangle \qquad \textsc{CondTrue}\ \frac{[\![e]\!](\mu) = \mathsf{true}}{\langle \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P', \mu\rangle \rightsquigarrow_1 \langle P, \mu\rangle} \qquad \textsc{CondFalse}\ \frac{[\![e]\!](\mu) \neq \mathsf{true}}{\langle \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P', \mu\rangle \rightsquigarrow_1 \langle P', \mu\rangle}$$

$$\textsc{WhileTrue}\ \frac{[\![e]\!](\mu) = \mathsf{true}}{\langle \mathsf{while}\ e\ \mathsf{do}\ P, \mu\rangle \rightsquigarrow_1 \langle P; \mathsf{while}\ e\ \mathsf{do}\ P, \mu\rangle} \qquad \textsc{WhileFalse}\ \frac{[\![e]\!](\mu) \neq \mathsf{true}}{\langle \mathsf{while}\ e\ \mathsf{do}\ P, \mu\rangle \rightsquigarrow_1 \langle \surd, \mu\rangle} \qquad \textsc{Fun}\ \frac{p = [\![f]\!](\mu(y_1), \ldots, \mu(y_n))(\vec{v}) \qquad p > 0}{\langle \vec{x} := f(y_1, \ldots, y_n), \mu\rangle \rightsquigarrow_p \langle \surd, \mu\{\vec{x} \mapsto \vec{v}\}\rangle}$$

Figure 4. Probabilistic operational semantics

and some variable $b \notin v(J, B_0, B_1, T)$ in the command

$$\begin{aligned} \mathit{CNI} \doteq\ & b := \{0,1\}; \\ & J; \mathsf{if}\ b = 0\ \mathsf{then}\ B_0\ \mathsf{else}\ B_1; \\ & P \end{aligned}$$

the advantage $|\Pr[\mathit{CNI}; T; b = g] - \frac{1}{2}|$ is negligible.

In the game of the definition, $J; \mathsf{if}\ b = 0\ \mathsf{then}\ B_0\ \mathsf{else}\ B_1$ probabilistically initialize variables. Then P runs. Finally, T attempts to guess the value of b and sets g accordingly. Hence, the property states that the two memory distributions for $b = 0$ and $b = 1$ after running P cannot be separated by an adversary that reads V. Semantically, this property can also be stated as indistinguishability (Mao 2003) of the ensembles d_b for $b = 0, 1$ defined by $\langle J; B_b; P, d_0(\eta)\rangle \Downarrow d_b(\eta)$ for the same range of commands for J and B_b. In case the program P is deterministic, this property is equivalent to non-interference (Definition 2): the adversary T guesses b correctly with probability $\frac{1}{2}$.

We finally generalize the property to account for active adversaries, as in Definition 7.

DEFINITION 12. (Computational non-interference against active adversaries). *Let P be a polynomial command context, Γ a policy for its variables, and $\alpha \in \mathcal{L}$.*

P is computationally non-interferent against α-adversaries when, for both $V, U = V_\alpha^C, \emptyset$ and $V, U = V_\alpha^I, V_\alpha^I \cap wv(P)$, and for all polynomial commands

- *J writing $V \setminus U$: $wv(J) \subseteq V \setminus U$;*
- *B_b for $b = 0, 1$ writing outside V: $wv(B_b) \cap V = \emptyset$;*
- *$\vec{A}$ α-adversaries;*
- *T reading V, writing g: $rv(T) \subseteq V$; $g \notin wv(J, B_0, B_1, \vec{A})$;*

and some variable $b \notin v(J, B_0, B_1, P, \vec{A}, T)$ in the command

$$\begin{aligned} \mathit{CNI} \doteq\ & b := \{0,1\}; \\ & J; \mathsf{if}\ b = 0\ \mathsf{then}\ B_0\ \mathsf{else}\ B_1; \\ & P[\vec{A}] \end{aligned}$$

if we have $\Pr[\mathit{CNI}; \bigwedge_{x \in rv(T)} x \neq \bot] = 1$, then the advantage $|\Pr[\mathit{CNI}; T; b = g] - \frac{1}{2}|$ is negligible.

The game of this definition performs initialization as in Definition 11, then runs $P[\vec{A}]$, and finally tests the resulting memory, as in Definition 7. The condition $\bigwedge_{x \in rv(T)} x \neq \bot$ prevents that T reads possibly-undefined memory. As can be expected, Definition 12 definition coincides with Definition 7 in the deterministic case.

3.4 Encryption

Our first cryptographic algorithms provide confidentiality by asymmetric (public-key) encryption. We represent them in our target language as three probabilistic functions $\mathcal{G}_e$, $\mathcal{E}$, and $\mathcal{D}$ that meet the functional and security properties given below.

DEFINITION 13 (Encryption scheme). *Let plaintexts, ciphertexts, publickeys, and secretkeys be sets of polynomially-bounded bitstrings indexed by η.*

An asymmetric encryption scheme is a triple of algorithms $(\mathcal{G}_e, \mathcal{E}, \mathcal{D})$ such that

- *$\mathcal{G}_e$, used for* key generation*, ranges over publickeys$\times$secretkeys;*
- *$\mathcal{E}$, used for* encryption*, ranges over ciphertexts;*
- *$\mathcal{D}$, used for* decryption*, ranges over plaintexts and is such that, for all $k_e, k_d := \mathcal{G}_e()$ and $m \in$ plaintexts, we have $\mathcal{D}(\mathcal{E}(m, k_e), k_d) = m$.*

The definition abstracts some details, such as input validation, or the possibility that decryption visibly fails on ill-formed inputs for instance. On the other hand, we need to specify (or at least bound) the set *plaintexts*, as we are going to require that encryption hides the length of plaintexts. (With our definition, the decryption of an encryption of an input outside *plaintexts* may fail, for instance when the input is too long, but at least the confidentiality of this input is still preserved.)

There are many different notions of security for encryption. The one we use is introduced by Rackoff and Simon (1991) and is the strongest usually considered; it can be realized under the Decisional Diffie-Hellman assumption. To code the definition in our target language, we rely on auxiliary primitive operations on lists: nil for the empty list, $+$ for concatenation, and $\in$ for membership test.

DEFINITION 14 (IND-CCA2 security). *Consider the commands*

$$\begin{aligned} E \doteq\ & \mathsf{if}\ b = 0\ \mathsf{then}\ m := \mathcal{E}(x_0, k_e)\ \mathsf{else}\ m := \mathcal{E}(x_1, k_e); \\ & \mathit{log} := \mathit{log} + m \\ D \doteq\ & \mathsf{if}\ m \in \mathit{log}\ \mathsf{then}\ x := 0\ \mathsf{else}\ x := \mathcal{D}(m, k_d) \\ \mathit{CCA} \doteq\ & b := \{0,1\}; \mathit{log} := \mathsf{nil}; k_e, k_d := \mathcal{G}_e(); A[E, D] \end{aligned}$$

The encryption scheme $(\mathcal{G}_e, \mathcal{E}, \mathcal{D})$ provides indistinguishability under adaptive chosen-ciphertext attacks when the advantage $|\Pr[\mathit{CCA}; b = g] - \frac{1}{2}|$ is negligible for any polynomial command context A with $b, k_d \notin rv(A)$ and $b, k_d, k_e, \eta, \mathit{log} \notin wv(A)$.

In this definition of security, *CCA* is a probabilistic command that represents a cryptographic game where the adversary is challenged to guess the secret bit b by interacting with an instance of the encryption scheme. The command A models an adversary that attempts to guess b as follows:

- A can perform arbitrary polynomial-time computation using any variables not excluded in the definition. For instance, A

may include commands that run the algorithms $\mathcal{G}_e$, $\mathcal{E}$, and $\mathcal{D}$ on any values that A can obtain or compute, including the encryption key k_e.

- A can also invoke encryption and decryption oracles, modelled as commands E and D, for any values of the parameters x_0, x_1, m, x, at any point in its code.

  (In the usual presentation of IND-CCA2, the adversary calls the encryption oracle E only once; however, the two definitions are equivalent, see Bellare et al. 2000.)

- A can set the variable g and terminate to report its guess of the value of the bit b.

In contrast with A, the commands E and D have access to the challenge bit b, the decryption key d, and the *log*. The encryption oracle E selects which of the two values stored in x_0 and x_1 to encrypt depending on b; it also maintains a log of encrypted values. The decryption oracle D provides decryption of any value except those produced by E.

For any run of the game, the adversary wins when $b = g$. The adversary $A \doteq g := \{0, 1\}$ wins with probability $\frac{1}{2}$, so the security property states that any adversary that meets our hypothesis cannot do (much) better, despite its control on the usage of the key.

The security definition we use assumes that the adversary provides all plaintexts to the encryption oracle. Hence, in particular, it does not cover more complex usages of encryptions, such as those where encrypted plaintexts may themselves depend on decryption keys. Said otherwise, IND-CCA2 says nothing about confidentiality in case the plaintexts may depend on the decryption key (see e.g. Abadi and Rogaway 2002). This situation is referred to as a key cycle, and will need to be excluded by typing.

For simplicity, we do not introduce primitives for symmetric (shared-key) encryption; their definition is similar, except that the adversary is not given access to the encryption key.

EXAMPLE 4. *Assume the adversary reads only x_{LH}, x'_{LH}, and k_e. The command*

$$k_e, k_d := \mathcal{G}_e(); x_{LH} := \mathcal{E}(y_{HH}, k_e); x'_{LH} := \mathcal{E}(0, k_e)$$

is computationally non-interferent (CNI). In particular, we have $x_{LH} = x'_{LH}$ with negligible probability, even if $y_{HH} = 0$, so any IND-CCA2 encryption function must be probabilistic.

Conversely, none of the three commands

$$P_1 \doteq x_{LH} := \mathcal{E}(0, y_{HH})$$

$$P_2 \doteq k_e, k_d := \mathcal{G}_e(); x_{LH} := \mathcal{E}(y_{HH}, k_e); x'_{LH} := \mathcal{E}(k_d, k_e)$$

$$P_3 \doteq k_e, k_d := \mathcal{G}_e(); k'_e, k'_d := \mathcal{G}_e(); \\ (\mathsf{if}\ y_{HH}\ \mathsf{then}\ k'_e := k_e); x_{LH} := \mathcal{E}(z_{HH}, k'_e)$$

is CNI for some IND-CCA2-secure encryption schemes: in the first case the encryption function is not properly used since y_{HH} is not a key; in the second command, there is a key cycle (k_e encrypts k_d) and IND-CCA2 does not give any assurance for encryptions of the decryption key; in the third command, key selection depends on a secret value, and IND-CCA2 does not prevent extracting k'_e from x_{LH} and comparing it with k_e.

EXAMPLE 5 (Confidentiality despite active adversaries). *Consider two commands mixing local secrets and adversary data, with shared access to keys k_e and k_d and low confidentiality variables x_0 and x_1.*

$$(P_u)_{u=0,1} = \mathsf{if}\ e_u = u\ \mathsf{then}\ h_u := \mathcal{D}(x_u, k_d); \\ s_u := h_u + l_u; \\ x_{1-u} := \mathcal{E}(s_u, k_e)$$

The command $k_e, k_d := \mathcal{G}_e(); \mathsf{while}\ e'\ \mathsf{do}\ (P_0; \_; P_1)$ does not leak any information on x_u, even if the adversary controls the values of e', e_u, l_u, x_u for $u = 0, 1$. (The command clearly does not protect the integrity on x_u.)

3.5 Cryptographic signatures

Our second cryptographic scheme provides integrity protection by asymmetric (public-key) signatures.

DEFINITION 15 (Signature scheme). *Let sigkeys, verifykeys, signedtexts, and plaintexts be sets of polynomially-bound bitstrings indexed by η.*

A signature scheme is a triple of algorithms $(\mathcal{G}_s, \mathcal{S}, \mathcal{V})$ such that

- *$\mathcal{G}_s$, used for* key generation*, ranges over sigkeys $\times$ verifykeys;*
- *$\mathcal{S}$, used for* signing*, ranges over signedtexts;*
- *$\mathcal{V}$, used for* signature verification*, ranges over $\{0, 1\}$ and is such that, for all $k_s, k_v := \mathcal{G}_s()$ and $m \in$ plaintexts, we have $\mathcal{V}(m, \mathcal{S}(m, k_s), k_v) = 1$.*

For convenience, we assume that $\mathcal{V}$ is deterministic, so that we can use test expressions $\mathcal{V}(e, e', e'')$ in conditional commands.

There are also many notions of security for signature schemes. We use a standard notion introduced by Goldwasser et al. (1988):

DEFINITION 16 (CMA security). *Consider the commands*

$$S \doteq x := \mathcal{S}(m, k_s); log := log + m;$$

$$CMA \doteq k_s, k_v := \mathcal{G}_s(); log := \mathsf{nil}; A[S]; \\ \mathsf{if}\ m \in log\ \mathsf{then}\ b := 0\ \mathsf{else}\ b := \mathcal{V}(m, x, k_v)$$

The signature scheme $(\mathcal{G}_s, \mathcal{S}, \mathcal{V})$ is secure against forgery under adaptive chosen-message attack when $\Pr[CMA; b = 1]$ is negligible for any polynomial command context A that cannot read k_s and cannot write k_s, k_v, log, η.

In the definition, command A represents an adversary that can

- invoke (as oracle) the command S for obtaining the signature x of any message m;
- read and write variables m, x; A may also run the verification algorithm, since it can access the verification key k_v;
- read but not write variables k_v, *log*, and η.

Conversely, A has no direct access to the signing key k_s. This game intuitively says that, after requesting as many signatures as he wants from the signing oracle S, the adversary still cannot produce a pair (m, x) such that x is the signature for a message m not signed by S, as recorded in *log*.

4. A type system for cryptography

We extend the type system of Section 2 to probabilistic programs, with special rules for typing the usage of cryptography.

In the rest of the paper, we assume given two fixed schemes for encryption and signing that meet Definitions 13, 14, 15 and 16, and such that, for each η, the sets *plaintexts* include all encrypted and signed values.

4.1 Types

We supplement the data type Data of Section 2 with types for cryptographic values. Data type safety is important for computational soundness inasmuch as the security of its primitives holds only when they are called with properly-generated keys, used only as keys. They also help prevent key cycles.

We use the following grammar for security types:

τ	::=	$t(\ell)$	Security types
t	::=	Data	Data types for payloads
	|	$\mathsf{Enc}\,\tau\,K \mid \mathsf{Ke}\,\tau\,K \mid \mathsf{Kd}\,\tau\,K$	Data types for encryption
	|	$\mathsf{Sig}\,\tau \mid \mathsf{Ks}\,\mathsf{F}\,K \mid \mathsf{Kv}\,\mathsf{F}\,K$	Data types for signing

$$\textsc{GenE}\quad \frac{\Gamma(k_e) = \mathsf{Ke}\,\tau K(\ell_e) \qquad \Gamma(k_d) = \mathsf{Kd}\,\tau K(\ell_d) \qquad C(\tau) \leq_C C(\ell_d)}{\vdash k_e, k_d := \mathcal{G}_e() \ : \ell_e \sqcap \ell_d}$$

$$\textsc{GenS}\quad \frac{\Gamma(k_s) = \mathsf{Ks}\,\mathsf{F}\,K(\ell_s) \qquad \Gamma(k_v) = \mathsf{Kv}\,\mathsf{F}\,K(\ell_v)}{\vdash k_s, k_v := \mathcal{G}_s() \ : \ell_s \sqcap \ell_v}$$

$$\textsc{Encrypt}\quad \frac{\vdash k_e : \mathsf{Ke}\,\tau K(\ell_x) \qquad \vdash y : \tau \qquad \Gamma(x) = \mathsf{Enc}\,\tau K(\ell_x) \qquad I(\tau) \leq_I I(x)}{\vdash x := \mathcal{E}(y, k_e) \ : \ell_x}$$

$$\textsc{Sign}\quad \frac{\Gamma(k_s) = \mathsf{Ks}\,\mathsf{F}\,K(\ell_s) \qquad \mathsf{F}(\mathsf{t}) = \tau \qquad \Gamma(x) = \mathsf{Sig}\,\tau(\ell_x) \qquad \vdash y : \tau \qquad L(\tau) \leq \ell_x \qquad I(\ell_s) \leq_I I(x)}{\vdash x := \mathcal{S}(\mathsf{t} + y, k_s) \ : \ell_x}$$

$$\textsc{Decrypt}\quad \frac{\Gamma(x) = \tau \qquad \vdash y : \mathsf{Enc}\,\tau K(L(x)) \qquad \vdash k_d : \mathsf{Kd}\,\tau K(L(x))}{\vdash x := \mathcal{D}(y, k_d) \ : L(x)}$$

$$\textsc{ProbFun}\quad \frac{\vdash y \ : \ \mathsf{Data}\,(\ell) \ \text{ for } y \in \vec{y} \qquad \mathsf{Data}\,(\ell) \leq \Gamma(x) \ \text{ for } x \in \vec{x}}{\Gamma \vdash \vec{x} := f(\vec{y}) \ : \ell}$$

$$\textsc{Verify}\quad \frac{\Gamma(k_v) = \mathsf{Kv}\,\mathsf{F}\,K(\ell_v) \qquad \mathsf{F}(\mathsf{t}) \leq \Gamma(x) \qquad \vdash \mathcal{V}(\mathsf{t} + y, m, k_v) : \mathsf{Data}\,(\ell') \qquad \vdash P : \ell_P \qquad C(\ell') \leq_C C(x) \sqcap C(\ell_P) \qquad I(\ell_v) \leq_I I(x)}{\vdash \mathsf{if}\ \mathcal{V}(\mathsf{t} + y, m, k_v)\ \mathsf{then}\ (x := y; P) \ : L(x) \sqcap \ell_P}$$

Figure 5. Typing rules for probabilistic commands with policy Γ.

where $\ell \in \mathcal{L}$ is a security label, K is a key label, and F is a map from tags to security types, as explained below.

Static key labels The labels K are used to keep track of keys, grouped by their key-generation commands. These labels are attached to the types of the generated key pairs, and propagated to the types of any derived cryptographic materials. They are used to match the usage of key pairs, to prevent key cycles, and to prevent generating multiple signatures with the same key and tag.

Tagged signatures Cryptographic signatures are often computed on (hashed) texts prefixed by a tag or some other descriptor that specializes the usage of the signing key. Accordingly, in order to precisely type expressions of the form $\mathcal{S}(\mathsf{t} + m, s)$ where t is a constant tag, our types for signing embed a partial map, F, from the tags usable with the key to the security type of the corresponding signed values. Otherwise, we would essentially have to use a distinct key for every signature.

EXAMPLE 6 (Tagged signatures). *The command context*

$$\begin{aligned} P[\_] \ \doteq\ & k_s, k_v := \mathcal{G}_s(); \\ & y_{LL} := \mathcal{S}(\mathsf{t0} + x_{LH}, k_s); z_{LL} := x_{LH}; \\ & y'_{LL} := \mathcal{S}(\mathsf{t1} + x'_{LH}, k_s); z'_{LL} := x'_{LH}; \\ & \_; \\ & \mathsf{if}\ \mathcal{V}(\mathsf{t0} + z_{LL}, y_{LL}, k_v)\ \mathsf{then}\ h_{LH} := z_{LL} \end{aligned}$$

is CNI (and typable) against an adversary that can read and write y_{LL}, y'_{LL}, z_{LL}, z'_{LL}. On the other hand, for the same class of adversaries, the command context obtained by erasing the two tags $\mathsf{t0}$ *and* $\mathsf{t1}$ *is not CNI for integrity, as can be seen for an adversary that overwrites the first value and signature with the second ones:*

$$\begin{aligned} J \ &\doteq x_{LH} := 0; x'_{LH} := 1 \\ B_b &\doteq z_{LL} := b \\ A \ &\doteq \mathsf{if}\ z_{LL}\ \mathsf{then}\ (y_{LL} := y'_{LL}; z_{LL} := z'_{LL}) \\ T \ &\doteq \mathsf{if}\ x_{LH} = x'_{LH}\ \mathsf{then}\ g := 0\ \mathsf{else}\ g := 1 \end{aligned}$$

Subtyping We rely on the two subtyping rules of the source type system, so subtyping between data types is just syntactic equality—we leave more interesting subtyping for future work. Note that subtyping from $\mathsf{Kd}\,\tau\,K$ to Data would not be sound in general, as it may hide some key dependencies and encryption cycles.

4.2 Typing rules

Our type system extends the source type system (Figures 1, 2, and 3) with the rules of Figure 5 for commands that call probabilistic functions, as explained below. It also has an additional rule for expressions, for typing signature verifications; this rule is identical to rule OP except for its cryptographic data types:

$$\textsc{OpVer}\quad \frac{\vdash y : t(\ell) \qquad \vdash m : \mathsf{Sig}\,t(\ell_s)(\ell) \qquad \vdash k_v : \mathsf{Kv}\,\mathsf{F}\,K(\ell) \qquad \mathsf{F}(\mathsf{t}) = t(\ell_s)}{\vdash \mathcal{V}(\mathsf{t} + y, m, k_v) \ : \ \mathsf{Data}\,(\ell)}$$

Rule PROBFUN is the generic rule for typing probabilistic functions; it requires that all variables have Data types and prevents explicit flows from the parameters $\vec{y}$ to the results $\vec{x}$. In particular, PROBFUN applies to the functions $\mathcal{S}$, $\mathcal{E}$, and $\mathcal{D}$ in case we do not rely on cryptographic assumptions. (The soundness of PROBFUN depends on the fact that f is a probabilistic function; side-effects in the evaluation of f would create correlation between successive calls to f.)

The rest of the rules are for cryptography; they permit some forms of declassification for encryptions (flows from higher to lower confidentiality levels) and endorsement for signature checks (flows from lower to higher integrity levels). The soundness of these rules depends both on cryptographic assumptions and on additional conditions on policies and programs, stated in Section 4.3.

GENS The two hypotheses bind key types to variables k_s and k_v, with the same map F from tags to payload types. The process label $\ell_s \sqcap \ell_v$ is the meet of the labels of all assigned variables.

SIGN Hypotheses 1, 3, and 4 bind types to the variables k_s, x, and y involved in signing; these types are related by τ, which sets the typing guarantees associated with signatures that use any signing key with key-label K and tag t.

Intuitively, we care mostly about the integrity of y (so that we only sign correct values) and the confidentiality of k_s (so that the adversary cannot sign incorrect values).

$L(\tau) \leq \ell_x$ records the flow from y to x, as in rule ASSIGN.

$I(\ell_s) \leq_I I(x)$ records the integrity flow from the signing key to the signature value. (Conversely, the confidentiality flow from k_s to x is ignored; this is sound only inasmuch as k_s is used only for signing.)

VERIFY The rule has a structure similar to rule CHECK. (Indeed, in the soundness proof, we use game rewritings to replace commands typable by VERIFY with commands typable by CHECK.) It also permits a limited form of endorsement: a lower integrity variable v can be assigned to a higher integrity variable x only if the guard performs a specific signature verification.

Hypotheses 1 and 2 check the verification-key type and relate the type $\mathsf{F}(\mathsf{t})$ for the tag used in the verification to the type of x.

The typing of the verification expression relies on rule OPVER; it records in ℓ' the ordinary flows from y, m, and k_v to the condition guard; it also enforces that y and $\mathsf{F}(\mathsf{t})$ (and thus x)

have identical data types. The typing of P records in ℓ_P the level of the guarded command.

The constraint on confidentiality levels records the implicit confidentiality flow from the guard to both the assignment and the guarded command. In contrast to normal assignment (ASSIGN), there is no constraint relating the integrity of ℓ' and x, as integrity follows from dynamic verification. Instead, an integrity constraint records a flow from the verification key to x.

GENE The hypotheses bind types to variables k_e and k_d with the same key-label and payload type τ—the type of plaintexts that can be encrypted and decrypted.

The constraint $C(\tau) \leq_C C(\ell_d)$ imposes the condition that confidentiality of the decryption key is greater or equal than the confidentiality of the plaintext.

ENCRYPT The first three hypotheses bind types to the variables k_e, y, and x involved in encryption; these types are related by τ and K, which describe the typing assumptions for encryption with key k_e.

The label ℓ_x in the typing of k_e records the flow from k_e to x (by subtyping, we have $L(k_e) \leq \ell_x$). The hypothesis $I(\tau) \leq_I I(x)$ records the integrity flow from y to x. Conversely, there is no constraint on the confidentiality flow from y to x, as encryption is a form of declassification: the rule is sound only with cryptographic assumptions.

DECRYPT The hypotheses bind types to the variables x, y, and k_d involved in decryption; these types are related by τ and K. The label $L(x)$ records flows from y and k_d to x, as in normal assignment.

Before stating our soundness result, we illustrate the type system on a few commands.

EXAMPLE 7 (Bad decrypted key). *Consider the command*

$$\begin{array}{l} k_e, k_d := \mathcal{G}_e(); k'_e, k'_d := \mathcal{G}_e(); \\ x_{LL} := \mathcal{E}(k'_e, k_e); \\ k_d := k'_d; \\ k := \mathcal{D}(x_{LL}, k_d); \\ y := \mathcal{E}(s, k) \end{array}$$

For the payload types $\tau' = \mathsf{Data}\,(\ell')$ and $\tau = \mathsf{Ke}\,\tau'\,K'(LH)$, we define Γ as follows:

$$\begin{array}{llll} \Gamma(k_e) & = \mathsf{Ke}\,\tau\,K(LH) & \Gamma(k_d) & = \mathsf{Kd}\,\tau\,K(HH) \\ \Gamma(k'_e) & = \tau & \Gamma(k'_d) & = \mathsf{Kd}\,\tau'\,K'(HH) \\ \Gamma(x_{LL}) & = \mathsf{Enc}\,\tau\,K(LL) & \Gamma(k) & = \tau \\ \Gamma(s) & = \tau' & \Gamma(y) & = \mathsf{Enc}\,\tau'\,K'(\ell) \end{array}$$

In case $C(\ell) \leq L$ and $C(\ell') \not\leq L$, this command is insecure because the key k_d used for decryption does not match the key k_e used for encryption. Hence, the decrypted value k is unspecified—it is unlikely to be a valid encryption key—and encryption using k is also unspecified—one can easily construct algorithms $\mathcal{G}_e$, $\mathcal{D}$, $\mathcal{E}$ that are IND-CCA2 and such that the final encryption leaks both its parameters s and k.

This command is not typable, since the data types of k_d and k'_d are not compatible.

This problem is not apparent in the work of Laud and Vene (2005) because, in their system, decryption never yield key types—the decrypted value k is just ordinary data that must remain secret irrespective of its distribution, so the final encryption is not typable. In our setting, we must address the problem in order to guarantee integrity as well as confidentiality after decryption. Although the program above is not typable either, we are able to type similar programs that rely on decrypted keys:

EXAMPLE 8 (Encrypt-then-sign an encryption key). *Consider the command context*

$$\begin{array}{l} k_e, k_d := \mathcal{G}_e(); k'_e, k'_d := \mathcal{G}_e(); \\ k_s, k_v := \mathcal{G}_s(); x_{LH} := \mathcal{E}(k'_e, k_e); \\ z_{LL} := \mathcal{S}(\mathsf{t} + x_{LH}, k_s); x_{LL} := x_{LH}; \\ \_; \\ \mathsf{if}\ \mathcal{V}(\mathsf{t} + x_{LL}, z_{LL}, k_v)\ \mathsf{then} \\ \quad (x'_{LH} := x_{LL}; k := \mathcal{D}(x'_{LH}, k_d); y := \mathcal{E}(s, k)) \end{array}$$

In this example, thanks to the signature verification, k is a valid decrypted key. If the adversary changes the signature stored in x, then signature verification fails, leaving m', k, and y uninitialized. The program is typable using the same Γ of Example 7 extended with types $\mathsf{Sig}\,(\mathsf{Enc}\,\tau\,K(LL))(LL)$ for z_{LL}, $\mathsf{Ks}\,\mathsf{F}\,K_s(LH)$ for k_s, $\mathsf{Kv}\,\mathsf{F}\,K_s(HH)$ for k_v, $\mathsf{Enc}\,\tau\,K(LH)$ for x'_{LH}, and $\mathsf{Enc}\,\tau\,K(LH)$ for x_{LH}, with $\mathsf{F}(\mathsf{t}) = \mathsf{Enc}\,\tau\,K(LH)$.

4.3 Computational soundness

We give additional conditions on policies and programs, then state our main soundness theorem. We require that the integrity of encryption keys is high enough for protecting confidentiality of plaintexts, and that the confidentiality of decryption keys is high enough for protecting integrity of signed values. These constraints relate integrity and confidentiality levels, depending on the capabilities of the adversary.

DEFINITION 17 (Robust policy). *Let Γ be a policy and $\alpha \in \mathcal{L}$.*

The encryption key type $\mathsf{Ke}\,\tau K(\ell)$ is robust at α when either $C(\tau) \leq_C C(\alpha)$ or $I(\ell) \leq_I I(\alpha)$.

The signing key type $\mathsf{Ks}\,\mathsf{F}\,K(\ell)$ is robust at α when either $I(\mathsf{F}(\mathsf{t})) \not\leq_I I(\alpha)$ for all $\mathsf{t} \in \mathsf{dom}(\mathsf{F})$ or $C(\ell) \not\leq_C C(\alpha)$.

The security-type policy Γ is robust at α when all its encryption and signing key types are robust at α.

In the two statements of robustness for key types, the first alternatives state that the protection provided by the key is irrelevant against an adversary at α, who could read plaintexts before encryption and write signed values before signing; the second alternatives demand that otherwise the key itself be sufficiently protected.

Besides the cryptographic assumptions, we state additional safety conditions for soundness.

DEFINITION 18. *A command context P is safe when $\Gamma \vdash P$ and*

1. *All commands guarded either by a signature verification or by a test typed by CHECK exclusively assign V^I_α in P; and P never reads uninitialized variables in V^I_α.*
2. *Each signing-key label/tag pair is used for signing at most once.*
3. *Each key label is used in at most one (dynamic) key generation.*
4. *Each key variable read in P is first initialized by P.*

These conditions are needed to apply the cryptographic games in the soundness proof of the target type system, for instance to guarantee the integrity of decrypted values. They can be enforced by static analysis, for instance by collecting all relevant static occurrences of variables and forbidding signing and encryption-key generation within loops.

Condition 1 helps deal with runtime errors, as discussed in Section 2. Condition 2 prevents signature replay attacks, as illustrated in Example 6. (We rely on a static key label so that the unique-signing constraint is shared between all aliases of any given signing key.) Condition 3 prevents decryption-key mismatches, as illustrated in Example 7. Condition 4 recalls our assumption on uninitialized variables for keys.

Relying on these conditions, we obtain that well-typed programs are computationally non-interferent (Definition 12).

THEOREM 4. *Let $\alpha \in \mathcal{L}$ be a security label. Let Γ be a policy that is robust at α. Let P be a safe polynomial-time command context.*

P satisfies computational non-interference against α-adversaries.

The proof relies on a series of typability-preserving program transformations that match the structure of the games used in the cryptographic security assumptions (Definitions 14 and 16). These transformations eliminate the cryptographic primitives, one static key label at a time. Hence, after eliminating a (static) keypair for signing and verification, the values that were signed and verified are now passed on auxiliary shared high-integrity variables, and we are left with conditionals typable by CHECK instead of VERIFY.

4.4 A memory-protection protocol

In preparation for our program translation, we define a protocol for sharing encrypted-then-signed memory. (Its typing assumptions are detailed in Section 5.)

$$\begin{aligned}
\mathit{Init}_s(k_s, k_v) &\doteq k_s, k_v := \mathcal{G}_s() \\
\mathit{Init}_e(k_e, k_d) &\doteq k_e, k_d := \mathcal{G}_e() \\
\mathit{Read}(x \leftarrow x_e, x_s, x'_e, k_d, k_v, \mathsf{t})[P] &\doteq \\
\quad \mathsf{if}\ \mathcal{V}(\mathsf{t} + x_e, x_s, k_v)\ \mathsf{then}\ (x'_e := x_e; x := \mathcal{D}(x'_e, k_d); P) & \\
\mathit{Write}(x_s, x_e \leftarrow x, x'_e, k_e, k_s, \mathsf{t}) &\doteq \\
\quad x'_e := \mathcal{E}(x, k_e); x_s := \mathcal{S}(\mathsf{t} + x'_e, k_s); x_e := x'_e &
\end{aligned}$$

Command Init_s generates keys for signing and verification. Command Init_e generates keys for encryption and decryption. Command context *Read* attempts to read x from x_e and x_s. It verifies x's presumed signature x_s, specifically for the tag t; if the verification succeeds, the ciphertext is copied to a temporary variable x'_e then decrypted to x, and finally command P runs. Otherwise the command silently fails. Conversely, command *Write* writes x to x_e and x_s, by first encrypting then signing x's value. (The temporary variables x'_e matter only for typing; they enable us to label x'_e with integrity higher than x_e.)

Depending on the relative levels of x and α, similar but simpler protocols may be used instead for protecting only confidentiality, or only integrity.

EXAMPLE 9 (Key Establishment). *Relying on the protocol above, we show how two hosts H and H' may dynamically establish new session keys using their long-term keys.*

We assume that variables with prefix H are local to H (readable and writable only in H) and variables with prefix H' local to H', respectively. We use the command context

$$\begin{aligned}
&\mathit{Init}_e(k_e^{H'}, H'.k_d^{H'}); \mathit{Init}_s(H.k_s^H, k_v^H); \\
&_; \\
&\mathit{Init}_e(H.k_e, H.k_d); \mathit{Init}_s(H.k_s, H.k_v); \\
&H.k := H.k_e + H.k_d + H.k_s + H.k_v; \\
&\mathit{Write}(x_s, x_e \leftarrow H.k, H.x_e, k_e^{H'}, H.k_s^H, t_k); \\
&_; \\
&\mathit{Read}(H'.k \leftarrow x_e, x_s, H'.x_e, H'.k_d^{H'}, k_v^H, t_k)[P']
\end{aligned}$$

The first command line models our hypothesis that H and H' share correctly-generated long-term keys ($k_e^{H'}$ and k_v^H), with a private signing key for H ($H.k_s^H$) and a private decryption key for H' ($H'.k_d^{H'}$). The third line represents session-key generation by H: four local keys are generated. These keys are then concatenated (in $H.k$) and sent (via shared memory x_e and x_s) towards H' using a Write command. The final line represents the reception of session keys by H'.

To type this command, we need a slight extension of our typing rules for concatenation. Alternatively, we can type a variant of this command where four runs of the Read/Write protocol are executed, one for each of the keys $H.k_e$, $H.k_d$, $H.k_s$, and $H.k_v$. On the other hand, one can optimize the protocol by avoiding encryption for the public keys k_e and k_v.

5. A cryptographic translation

We are now ready to describe and verify general cryptographic protection mechanisms for shared memory. To illustrate our type-based approach, we provide a simple translation from programs that rely on shared-memory security policies to programs that rely on cryptography.

Modelling distributed systems We model distributed systems as series of commands from the source language located at different hosts that communicate through shared memory. Untrusted shared memory may conservatively represent a public network, or a protocol stack for example.

We consider source systems of the form $P_1; \ldots; P_n$, such that the control flow between threads is statically known. We let $\vec{h}$ be the set of all host names. Each host may have several successive threads, which may share private state using the host's local memory. Thus, we define (source) systems as sequences of threads

$$S \ ::= \ (P)^i \ \mid \ S; S$$

where each thread $(P)^i$ is annotated with a unique thread identifier i. We let γ be a mapping from thread identifiers to hosts. (Holes do not appear in source systems; the translation insert them between translated threads.)

We assume given a set of variables, X, shared between some of these hosts, that require cryptographic protection. To obtain a realistic distributed implementation, this set X should contain all variables shared between any two hosts, except possibly for some initialization variables.

We also assume that every occurrence of every variable $x \in X$ in an expression of S is correctly annotated by its last-writer-thread: we write x^i when i is (always) the last thread to have written x when the expression gets evaluated. If there is no such thread (in particular, if a variable of X may be read before being written in S), the program cannot be correctly annotated. These annotations can be inferred (or checked) using conservative static analyses. In the translation, they help us meet the requirements on signatures, and thus prevent replay attacks, as we always know which verification key and tag should be used when x is read.

In summary, the source inputs of the translation consist of a policy Γ, a subset X of its domain indicating the variables to protect, and a well-typed, correctly annotated system S.

Although our translation takes as input systems, i.e. annotated commands, the annotations do not affect source command typing: after erasing thread- and writer-annotations, we use the simple type system of Section 2.4, Figures 1 and 2.

Public-key infrastructure We assume given a public-key infrastructure for signature verification keys: every host h has a signing key k_s^h and knows every other host's verification key k_v^h. We initialize these keys by running the command

$$\mathit{Init}\ \vec{h} = (k_s^h, k_v^h := \mathcal{G}_s();)_{h \in \vec{h}}$$

Auxiliary variables The translation uses the following naming conventions and types for variables. We assume that the variables introduced by the translation do not occur in the source system and are pairwise distinct.

To every source variable $x \in X$, the translation associates two shared variables, x_e containing the encrypted value of x, and x_s containing the signature for x_e. In addition, for every thread i that accesses x in the source program, the translation uses a series of local variables: $i.x$ is a local copy of x; $i.x_e$ and $i.x'_e$ are local

buffers for the encrypted value of x; $i.x_v$ is a local buffer for x after verification.

Finally, the translation uses two functions from $x \in X$ to variables holding encryption and decryption keys: $k_e(x)$ is the encryption key for writing x; $k_d(x)$ is the decryption key for reading x. We do not assume that these functions are injective: on the contrary, subject to typing constraints, the same keys should be used to protect many shared variables. We initialize all variables in the range of these functions by running the command

$$Init_X = (k_d(x), k_e(x) := \mathcal{G}_e();)_{x \in X}$$

where each key pair $k_e(x), k_d(x)$ for $x \in X$ is initialized just once.

Translating security policies We now translate Γ, thereby giving types to all target variables: we let $[\![\Gamma]\!]$ be the security policy that coincides with Γ on $\mathsf{dom}(\Gamma) \setminus X$ and maps the translation variables to the types given below.

For simplicity, we use a single security label for all unprotected shared memory (after encryption and signing): we let ℓ' be a security level such that $I(\ell') = I(\top^{\mathcal{L}})$ and $C(\ell') = C(\bot^{\mathcal{L}})$. (Alternatively, we could parameterize the translation with target security labels, and write a set of constraints to be satisfied in order to preserve typability.)

For every host $h \in \vec{h}$, we let

$$[\![\Gamma]\!](k_s^h) = \mathsf{Ks}\ \mathsf{F}\ K_h(\ell_s^h) \qquad [\![\Gamma]\!](k_v^h) = \mathsf{Kv}\ \mathsf{F}\ K_h(\ell_v^h)$$

where K_h is a unique static key label (no other host will have the same label), F is a mapping from tags to security types, and ℓ_s^h, ℓ_v^h are security labels, subject to the constraints given below.

For every $x \in X$, let $\tau = \Gamma(x)$. Since source types do not use cryptography, we have $\tau = \mathsf{Data}\ (\ell)$ for some ℓ. Let $\ell_e \in \mathcal{L}$ such that $I(\ell_e) = I(\ell)$ and $C(\ell_e) = \bot_C$. We let

$$\begin{array}{ll}
[\![\Gamma]\!](x_e) & = \mathsf{Enc}\ \tau\ K_x\ (\ell') \\
[\![\Gamma]\!](x_s) & = \mathsf{Sig}\ \mathsf{Enc}\ \tau\ K_x(\ell_e)\ (\ell') \\
[\![\Gamma]\!](k_e(x)) & = \mathsf{Ke}\ \tau\ K_x(\ell_e) \\
[\![\Gamma]\!](k_d(x)) & = \mathsf{Kd}\ \tau\ K_x(\ell_d) \mid C(\ell) = C(\ell_d) \wedge I(\ell_d) \leq_I I(\ell)
\end{array}$$

where K_x is the static, unique key label for the keypair $k_e(x), k_d(x)$ (no other key will have the same label).

For every thread i that accesses x, we let

$$\begin{array}{ll}
[\![\Gamma]\!](i.x) = \tau & [\![\Gamma]\!](i.x_e) = \mathsf{Enc}\ \tau\ K_x(\ell_e) \\
[\![\Gamma]\!](i.x_v) = \tau & [\![\Gamma]\!](i.x'_e) = \mathsf{Enc}\ \tau\ K_x(\ell_e)
\end{array}$$

Moreover, if i writes x at host $h = \gamma(i)$, we have three constraints

$$\ell_v^h \leq \ell_e \qquad I(\ell_s^h) \leq_I I(\ell_e) \qquad \mathsf{F}(\text{`}x\text{'} + i) = \mathsf{Enc}\ \tau\ K_x(\ell_e)$$

Translating systems and threads We now translate commands, relying on the command definitions and notations of the memory protocol of Section 4.4. Our translation protects every thread $(P_i)^i$ by first decoding every shared variable of X (syntactically) read by P_i into local memory then, if all verifications succeed, running (a variant of) P_i on local memory and, finally, committing every (syntactically) written variable of X back to shared encrypted memory.

For every thread $(P_i)^i$ of S, we let $R^i[\_]$ be the composition of command contexts

$$\mathit{Read}(i.x_v \leftarrow x_e, x_s, i.x_e, k_d(x), k_v^{\gamma(w)}, \text{`}x\text{'} + w)[\_]$$

for all annotated variables x^w in $rv(P_i) \cap X$ with $w \neq i$, let P'_i be the sequence of assignments $i.x := i.x_v$ for the variables $i.x_v$ assigned in R^i, and let W^i be the sequential composition of commands

$$\mathit{Write}(x_s, x_e \leftarrow i.x, i.x'_e, k_e(x), k_s^{\gamma(i)}, \text{`}x\text{'} + i)$$

for all $x \in wv(P_i) \cap X$. We arrive at the top-level translation:

$$[\![P_i]\!]^i = R^i\left[P'_i; P_i\{i.x/x \text{ for } x \in X\}; W^i\right]$$

$$[\![(P_0)^0; \ldots; (P_n)^n]\!] = \mathit{Init}\ \vec{h}; \mathit{Init}_X; \_; [\![P_0]\!]^0; \_; \ldots; \_; [\![P_n]\!]^n; \_$$

For each translated thread, each variable in X is written at most once. This ensures that each tag pair (composed by a unique thread identifier and a variable name) is used at most once. Moreover, the auxiliary assignments P'_i ensure that the local variables $i.x$ are assigned only if all verifications succeed; they guarantee exclusive assignments for these variables.

EXAMPLE 10. *In the 4-point lattice, consider the translation of*

$$(y_{HH} := 0)^0; (\mathsf{if}\ y_{HH}^0 = 0\ \mathsf{then}\ x_{HH} := 1)^1$$

with two threads located at hosts h_0 and h_1 that share a single key pair k_e, k_d for all encryptions. For $X = \{y_{HH}\}$, the two translated threads are:

$$\begin{array}{l}
P_0 = 0.y_{HH} := 0; 0.y_{LH,e} := \mathcal{E}(0.y_{HH}, k_e); \\
\qquad y_{LL,s} := \mathcal{S}(\text{`}y_{HH}\text{'} + 0 + 0.y_{LH,e}, k_s^{h_0}); y_{LL,e} := 0.y_{LH,e} \\
P_1 = \mathsf{if}\ \mathcal{V}(\text{`}y_{HH}\text{'} + 0 + 0.y_{LL,e}, y_{LL,s}, k_v^{h_0}) \\
\qquad \mathsf{then}(1.y_{LH,e} := y_{LL,e}; 1.y_{HH,v} := \mathcal{D}(1.y_{LH,e}, k_d); \\
\qquad\quad 1.y_{HH} := 1.y_{HH,v}; \mathsf{if}\ 1.y_{HH} = 0\ \mathsf{then}\ x_{HH} := 1)
\end{array}$$

Discussion The translation systematically performs two cryptographic operations for every access to a shared variable in X. Still, we believe that our type system also supports a variety of other options, which would be selected by more advanced translations. As illustrated in Example 9, the translation may reduce cryptographic costs by clustering variables with the same level into fewer, larger shared variables (representing messages, or pages in a distributed file system, for instance). Besides, depending on source labels, the translation may select simpler read and write protocol that omit encryption or signing for low-integrity or low-confidentiality source variables. Using more expressive types for cryptographic payloads, the translation may also jointly sign several values. For instance, a single signature suffices after executing every thread, and may be typable by associating a constant tag to a series of authenticated types, for a fixed series of variable writes.

When reading or writing in a high-confidentiality command, we cannot let runs of the read or write protocols be observable. To avoid this, the translation pre-fetches all variables potentially read under high-confidentiality guards, and rewrite all variables potentially written under a high-confidentiality guard. Consider for instance the second source thread of Example 10, in case x_{HH} is added to X. If we directly translated the update $x_{HH} := 1$ by a run of the write protocol under the high-confidentiality guard, an adversary would be able to compare ciphertexts x_e before and after running the thread, and thus infer the value of y_{HH} by observing whether the second thread updates the encrypted value or not. In our translation (with $x_{HH} \in X$), the local value $1.x$ is always re-encrypted, irrespective of y_{HH}, and thus always looks opaque and different from its previous value to a polynomial adversary.

Correctness We first verify that, in the absence of an active adversary, our translation is functionally correct, that is, the translated system always terminates with the same results as the source system. (The translated memory also includes cryptographic materials; the domain condition in the statement below excludes their values.)

For a given source configuration $\langle P, \mu \rangle$ such that μ is uninitialized on X, we let μ_d be the memory defined as follows: for every host $h \in \vec{h}$, and for every $x \in X$ and thread i of P such that x occurs in i, k_s^h, k_v^h, x_e, x_s, $k_e(x)$, $k_d(x)$, $i.x$, $i.x_v$, $i.x_e$, and $i.x'_e$ are defined and uninitialized; for every $x \in \mathsf{dom}(\mu) \setminus X$, $\mu_d(x) = \mu(x)$; and we let $[\![\langle P, \mu \rangle]\!]$ be the distribution of configurations such that $[\![\langle P, \mu \rangle]\!](P, \mu_d) = 1$.

THEOREM 5 (Functional Adequacy). *Let S be a correctly-annotated source system and μ a memory such that $\langle S, \mu\rangle \rightsquigarrow^* \langle\surd, \mu'\rangle$.*

We have $[\![\langle S[\vec{\mathsf{skip}}], \mu\rangle]\!] \rightsquigarrow^* d_T$ for a distribution d_T that gives probability 1 to configurations $\langle\surd, \mu'_T\rangle$ such that μ'_T coincides with μ' on their joint domain.

In the theorem, the distribution of final states d_T may give positive probabilities to a range of different memories μ'_T, with for instance different keys and different encrypted values, but these memories at least coincide on source variables outside X. (We can obtain adequacy for the final values of source variables in X by copying them to variables outside X at the end of P.) Also, since μ is fixed, we do not need to assume that P is polynomial: for a given run, we can select sets *plaintexts* for Definitions 13 and 15 that include at least the values encrypted and signed in this run.

We now consider the security of the translation, under the assumption that the source system is well-typed.

THEOREM 6 (Typability Preservation). *Let S be a correctly annotated source system. If $\Gamma \vdash S$, then $[\![\Gamma]\!] \vdash [\![S]\!]$.*

By computational soundness of typing (Theorem 4), we obtain:

THEOREM 7 (Computational soundness for the translation). *Let $\alpha \in \mathcal{L}$ a security label, Γ a source security policy, $X \subseteq \mathsf{dom}(\Gamma)$, and $S = (P_0)^0; \ldots; (P_n)^n$ a correctly-annotated source system such that $\Gamma \vdash S$ and $P_0, \ldots, P_n$ exclusively assign $V_\alpha^I \setminus X$.*

If $[\![\Gamma]\!]$ is a robust policy at α, then $[\![S]\!]$ is computationally non-interferent against α-adversaries.

6. Conclusions and future work

We presented a cryptographic type system for verifying the correct usage of encryption and signing schemes in programs with information-flow security policies. and used it to develop a secure translation for imperative programs that share protected memory. Security is defined by a computational non-interference property against active probabilistic, polynomial adversaries.

We would like to extend both the type system and the translation to obtain more efficient implementations for a larger class of programs (for instance by using dependent types rather than tags for signing keys, by extending the subtyping relation, and by supporting symmetric-key cryptography). Independently, it would be interesting to extend the class of security properties preserved by cryptographic implementations, for instance to account for controlled forms of declassifications and endorsements in source programs.

Acknowledgments

We thank Ricardo Corin, James Leifer, Jean-Jacques Lévy, Andrei Sabelfeld, Nobuko Yoshida, and the anonymous reviewers for their helpful comments.

References

Martín Abadi. Protection in programming-language translations. In *25th International Colloquium on Automata, Languages and Programming*, volume 1443 of *LNCS*, pages 868–883. Springer-Verlag, 1998.

Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

Martín Abadi, Ricardo Corin, and Cédric Fournet. Computational secrecy by typing for the pi calculus. In *APLAS'06*, volume 4279 of *LNCS*, pages 253–269. Springer-Verlag, November 2006.

Pedro Adão and Cédric Fournet. Cryptographically sound implementations for communicating processes (extended abstract). In *33rd International Colloquium on Automata, Languages and Programming*, volume 4052 of *LNCS*, pages 83–94. Springer-Verlag, July 2006.

Aslan Askarov, Daniel Hedin, and Andrei Sabelfeld. Cryptographically-masked flows. In *Proceedings of the 13th International Static Analysis Symposium*, LNCS, Seoul, Korea, 2006. Springer-Verlag.

M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations. In *10th ACM Conference on Computer and Communications Security*, pages 220–230, 2003.

Michael Backes. Quantifying probabilistic information flow in computational reactive systems. In *ESORICS'05*, volume 3679 of *LNCS*, pages 336–354. Springer-Verlag, September 2005.

Mihir Bellare and Phillip Rogaway. The game-playing technique, December 2004. At `http://www.cs.ucdavis.edu/~rogaway/papers/games.html`.

Mihir Bellare, Alexandra Boldyreva, and Silvio Micali. Public-key encryption in a multi-user setting : Security proofs and improvements. In *EUROCRYPT*, pages 259–274, 2000.

Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

Shafi Goldwasser, Silvio Micali, and Ronald Rivest. A digital signature scheme secure against adaptive chosen-message attack. *SIAM Journal on Computing*, 17(2):281–308, 1988.

Holger Hermanns. *Interactive Markov Chains: The Quest for Quantified Quality*. Springer Berlin/Heidelberg, 2002.

Peeter Laud. Secrecy types for a simulatable cryptographic library. In *12th ACM Conference on Computer and Communications Security*, pages 26–35, 2005.

Peeter Laud. Semantics and program analysis of computationally secure information flow. In *10th European Symposium on Programming (ESOP 2001)*, volume 2028 of *LNCS*. Springer-Verlag, April 2001.

Peeter Laud. On the computational soundness of cryptographically-masked flows. In *Proceedings of the 35th Symposium on Principles of Programming Languages*, San Francisco, USA, 2008. ACM Press.

Peeter Laud and Varmo Vene. A type system for computationally secure information flow. In *Fundamentals of Computation Theory*, LNCS, pages 365–377. Springer-Verlag, 2005.

Wenbo Mao. *Modern Cryptography: Theory and Practice*. Prentice Hall Professional Technical Reference, 2003.

David Monniaux. *Analyse de programmes probabilistes par interprétation abstraite*. PhD thesis, Université Paris IX Dauphine, 2001.

Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.

Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.

Charles Rackoff and Daniel R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *CRYPTO'91*, volume 576 of *LNCS*, pages 433–444. Springer-Verlag, 1991.

Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

Geoffrey Smith and Rafael Alpízar. Secure information flow with random assignment and encryption. In *FMSE '06: Proceedings of the fourth ACM workshop on Formal methods in security*, pages 33–44, 2006.

Jeffrey A. Vaughan and Steve Zdancewic. A cryptographic decentralized label model. In *IEEE Symposium on Security and Privacy*, pages 192–206, May 2007.

Steve Zdancewic and Andrew Myers. Robust declassification. In *14th IEEE Computer Security Foundations Workshop*, pages 15–23, 2001.

Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20(3): 283–328, 2002.

Lantian Zheng, Steve Chong, Andrew Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *15th IEEE Symposium on Security and Privacy*, 2003.