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, and we develop a type system for the target language. Our typing 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 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

\[(x := 1)^a; (if \ x \ then \ y := 2 \ else \ y := z)^b; (y := y + 1)^a;\]

where the parentheses \( (\cdot)^a \) and \( (\cdot)^b \) indicate code that runs on behalf of a and b, respectively, and where the placeholders \( _{\cdot} \) 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.

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 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright © 2008 ACM 978-1-59593-689-9/08/0001 . . . $5.00
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 \(x_e, x_s, y_e, y_s\) are used to pass the encrypted values and signatures for \(x\) and \(y\), respectively, the second command (if \(x\) then \(y := 2\) else \(y := z\)) may be implemented as

\[
\begin{align*}
\text{if } V(x_e, x_s, k_e) \text{ then } & \quad x'^b := D(x_e, k_d); \\
\text{if } x'^b \text{ then } & \quad y'^b := 2 \text{ else } y'^b := z'^b; \\
y_e := E(y'^b, k_s); & \quad y_s := S(y_e, k_s)
\end{align*}
\]

where, in order to read \(x\) out of its wire format \(x_e, x_s\), the code first verifies \(V\) the signature then performs a decryption \(D\) to extract a local copy of \(x\) into \(x'^b\); and, conversely, for writing \(y\), the code first encrypts \(E\) its updated copy \(y'^b\) and then signs \(S\) 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 \(k_e\) 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 \(y_s := x_e; y_s := x_s\) between \(b\) and \(o\). 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 illegitimately 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 graphically masked flows for non-deterministic semantics of encryption. Recently, Laud (2008) investigates conditions such that graphically 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.


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:

\[
\begin{align*}
e & ::= x \mid v \mid op(e_1, \ldots, e_n) \\
P & ::= e \mid P \mid P \mid \text{if } e \text{ then } P \text{ else } P \mid \text{while } e \text{ do } P \mid \text{skip}
\end{align*}
\]

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

We let \(\mu\) range over memories, that is, finite functions from variables to values plus a special term \(\bot\). We denote by \(\mu(x) = \bot\) that variable \(x\) is not initialized in memory \(\mu\). We assume that \(\bot\) 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 \(\mu_{\bot}\) be the memory that maps every variable to \(\bot\).

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 \((P, \mu)\), plus inert configurations, written \((\emptyset, \mu)\) or just \(\mu\), that represent command termination with final memory \(\mu\). We implicitly assume that all the variables of \(P\) are in the domain of \(\mu\). We let \(\sim\) represent single-step execution of commands between configurations and let \(\sim^*\) be its reflexive transitive closure. We denote normal termination as \((P, \mu) \sim^* \mu'\), that is, starting with initial memory \(\mu\), command \(P\) completes with final memory \(\mu'\) 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 \((\mathcal{L}_i, \leq)\). We write \(\mu \sqsubseteq \tau\) for the smallest and largest elements of \(\mathcal{L}\), and \(\mathcal{L}\) 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 \(\ell_c\) specifies a read level for variables, while the integrity level \(\ell_i\) specifies a write level; the meaning of \(\ell \leq \ell'\) is that \(\ell'\) is more confidential (can be read by fewer commands) and less integral (can be written by more commands) than \(\ell\). 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 \(C(\ell) \leq C(\ell')\) and \(I(\ell) \leq I(\ell')\). 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 \(\Gamma\) from variables to security types \(\tau\) of the form \(\ell(\ell)\), where \(\ell\) (for now) can only be instantiated with data type \(\mathcal{D}\) and where \(\ell\) is a security label. Overloading our notations, we lift our confidentiality and integrity projections \(C\) and \(I\) from labels to security types, and let \(\Gamma(\ell(\ell)) = C(\ell)\) (resp. \((I(\ell(\ell)) = I(\ell))\). We also let \(T(\ell(\ell)) = \ell\). 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 \(\mu_0\) and \(\mu_1\) are indistinguishable on \(V\), written \(\mu_0 \sim_{\mathcal{V}} \mu_1\), 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 \(\mu_0\) and \(\mu_1\), if \(\mu_0 \sim_{\mathcal{V}} \mu_1\) and \((P, \mu_0) \not\rightarrow \mu'\) for \(b = 0, 1\), then \(\mu'_0 \sim_{\mathcal{V}} \mu'_1\).

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 $\alpha$, passive case). Let $\Gamma$ be a memory policy and $\alpha \in L$ a security label. Let

$$V^G_\alpha = \{x \mid C(\Gamma(x)) \leq C(\alpha)\} \quad V'_\alpha = \{x \mid I(\Gamma(x)) \leq I(\alpha)\}$$

The command $P$ preserves confidentiality at $\alpha$ when it is non-interferent on $V^G_\alpha$; it preserves integrity at $\alpha$ when it is non-interferent on $V'_\alpha$.

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

$$rw(A) \subseteq V^G_\alpha \quad wv(A) \cap V'_\alpha = \emptyset$$

In the rest of the paper, we let $\alpha$ denote the security level of the adversary. As usual, once $\alpha$ is fixed, we may restrict our attention to the product of binary lattices $L \subseteq H$ for confidentiality and $H \subseteq L$ 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 $\alpha$.

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

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 $P[\mathcal{A}[P]]$ 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

$$P[\mathcal{A}[Q]] \quad l := 4; \quad \text{if } l = 4 \text{ then } h := 10 \text{ else } Q$$

where the hole $\mathcal{A}$ stands for low-level code. After running command $P[\text{skip}; h := 5]$ 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 $P[\text{skip}, \text{skip}]$, 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 $P[\mathcal{A}[\mathcal{B}]]$, 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 $P[l := 0, \text{skip}]$ (starting with memory $l \leftarrow \perp, h \leftarrow \perp$) leaves $h$ uninitialized. Further, we assume that programs never read variables that are uninitialized and not writable by the adversary ($I(x) \leq I(\alpha)$). 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 $\mu_0$ and $\mu_1$ are weakly indistinguishable on $V$, written $\mu_0 \sim_{\mathcal{V}} \mu_1$, when $x \in V$ implies either $\mu_0(x) = \mu_1(x)$, or $\mu_0(x) = \perp$, or $\mu_1(x) = \perp$.

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 $\mu_0$ and $\mu_1$, if

1. $\mu_0(x) = \mu_1(x) = \perp$ for every $x \in U$,
2. $\mu_0 \sim_{\mathcal{V}} \mu_1$, and
3. $\{\mu_0, \mu_1\} \downarrow \mu_0''$ for $b = 0, 1$,

then $\mu_0'' \sim_{\mathcal{V}} \mu_1''$.

**Definition 6** (Weak non-interference at $\alpha$). Let $\Gamma$ be a memory policy and $\alpha \in L$ a security label. The command $P$ is weakly non-interferent at $\alpha$ when $P$ is weakly non-interferent on both $V^G_\alpha, \emptyset$ and $V'_\alpha, V'_\alpha \cap \text{wv}(P)$.

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 $\Gamma$ be a memory policy and $\alpha \in L$ a security label.

The command context $P$ is non-interferent against $\alpha$-adversaries when, for both $V, U = V^G_\alpha, \emptyset$ and $V, U = V'_\alpha, V'_\alpha \cap \text{wv}(P)$, and for all commands

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

the value of $g$ after running $J$; $B_0: P[\overline{A}]; T$ does not depend on $b$: if $\langle J; B_0; P[\overline{A}]; T \rangle$ then $P[\overline{A}; \mu_\perp] \uparrow \mu'_\perp, P[\alpha \Downarrow_{\mathcal{V}} \mu_0''(x)] \neq \perp$, and $(T; \mu_0'') \uparrow \mu_0''$ for $b = 0, 1$ then we have $\mu_0''(g) = \mu_1''(g)$. 

The command \( G_b = J; B_0; P[A]; T \) represents a game, parameterized by a meta-variable \( b \) that is either 0 or 1; its first part \( J; B_0 \) initializes variables and, depending on \( b \), yields two indistinguishable memories (\( \mu_0 \) and \( \mu_1 \) in Definition 2); the second part \( P[A] \) runs the command context \( P \) in combination with active adversaries \( \hat{A} \); this yields two memories (\( \mu_0 \) and \( \mu_1 \) in Definition 2); the final part of the command, \( T \), represents an observer that attempts to guess the value of \( b \) from memory \( \mu_0 \). Intuitively, an opponent player that chooses the commands \( J, B_0, B_1, \hat{A}, T \) wins when \( g = 0 \) after running \( G_0 \) and \( g = 1 \) after running \( G_1 \).

**Example 1** (Non-interference for a simple lattice). Let \( L \) be the 4-point lattice with labels \( HL, HH, LL, LH \). Let \( \alpha = LH \). Let \( P \) be a command context that writes \( wv(P) = \{ x_{HL}, x_{LL}, x_{IL}, x'_{IL} \} \) and reads \( rv(P) = wv(P) \cup \{ x_{HL} \} \), with a single hole. To show that \( P \) is non-interferent against \( \alpha \)-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 \( x_{HL}, x_{IL} \) and \( x'_{IL} \);
  - \( B_0 \) sets \( x_{HL} \) and \( x_{IL} \) depending on \( x_{HL}, x_{IL} \) and \( x'_{IL} \);
  - \( A \) sets \( x_{HL} \) and \( x_{IL} \) depending on \( x_{HL}, x_{HL} \) and \( x'_{IL} \);
  - \( T \) sets \( g \) depending on \( x_{HL}, x_{HL}, x_{HL} \) and \( x'_{IL} \);

- for the integrity game:
  - \( J \) sets \( x_{HL} \);
  - \( B_0 \) sets \( x_{HL} \) and \( x_{IL} \) depending on \( x_{HL} \);
  - \( A \) sets \( x_{HL} \) and \( x_{IL} \) depending on \( x_{HL}, x_{HL}, x_{HL} \);
  - \( T \) sets \( g \) depending on \( x_{HL} \) and, only when they are initialized, on \( x_{HL} \) and \( x_{HL} \).

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

\[
J = \text{skip} \quad A = z_{HL} := y_{HL} \quad B_0 = x_{HL} := b \quad T = g := z_{HL}
\]

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 \( \alpha \) if and only if \( P \) (as a command context \( P \) with no \( \tau \)) is non-interferent against active adversaries.

In the general case, when \( P \) has at least one hole, non-interference against active adversaries implies that \( P[\text{skip}] \) 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

\[
P = x_{HL} := 0; \text{if } x_{HL} = 4 \text{ then } y_{HL} := 1 \text{ else } y_{HL} := 0
\]

\( P[\text{skip}] \) is weakly non-interferent, but \( P[\tau] \) is interferent against active adversaries for the integrity game, since we can define

\[
J = \text{skip} \quad A = \text{if } x_{HL} = 1 \text{ then } x_{HL} := 4
\]

\( B_0 = z_{HL} := b \quad T = g := y_{HL} \)

### 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 against active adversaries (Definition 7). The resulting type system is further extended in Section 4 to account for cryptographic primitives.

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 set \( t(\ell) \leq t'(\ell') \) iff \( \ell = \ell' \) and \( \ell \leq \ell' \).

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

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 \( \Gamma \) be a security policy and \( \alpha \in \mathcal{L} \) a security label. If \( \Gamma \vdash P : \ell \), then \( P \) is non-interferent at \( \alpha \).

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

Rule \textbf{HOLE} types placeholders for \( \alpha \)-adversaries with \( \tau \). Although the rule does not enforce any restriction, Definition 7 allows only \( \alpha \)-adversaries to be placed in holes.

Rule \textbf{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 \( M \) 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, \( P = \sum_i P_i \) for some command context \( Q_i \), for \( i = 1 \ldots n \).

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

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

**Theorem 3.** Let \( \Gamma \) be a policy and \( \alpha \in L \) a security label. Assume \( \Gamma \vdash P \) and all commands \( P' \) that occur in commands if \( \eta \) then \( P' \) typed by \( \text{CHECK} \) exclusively assign \( V \) in \( P \).

Then \( P \) is non-interferent against active \( \alpha \)-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 \( f \).

\[
P := \ldots \mid x_1, \ldots, x_m := f(y_1, \ldots, y_n)
\]

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

Every function \( f \) is equipped with an associated parametric probability distribution \( \mathbb{P}[f] \). (Hence, in the special case \( f \) is a deterministic function, the distribution \( \mathbb{P}[f] \) gives probability 1 to \( f(x) \) and 0 to any other output.) We write \( \{0,1\}^\infty \) for the “coin-tossing” function that returns either 0 or 1 with probability \( \frac{1}{2} \).

Instead of single configurations \((P, \nu)\), we now consider distributions of configurations, ranged over by \( \delta \). The operational semantics for commands is given in Figure 4, as a Markov chain (Hermanns 2002) on the set of all configurations, written \( \mathcal{S} \), with probabilistic steps induced by the function distributions. (Formally, rule \text{STATE} guarantees that all states have leaving transitions whose probabilities sum to 1.) We omit the usual semantics for expressions \( \{e\}(\nu) \).

An initial distribution for \( P \) is a distribution of configurations that has all its weight on configurations of the form \((P, \nu)\). 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 \( \text{prob} : \mathcal{S} \times \mathbb{S} \to [0,1] \) such that \( \mathcal{S} \sim_{\text{prob}} (s, s') \) in Figure 4 and \( T(d) \) be the distribution transformer such that

\[
T(d) \sim_{\text{prob}} (s') = \sum_{s' \in \mathcal{S}} \text{prob}(s, s') d(s)
\]

The semantics of a probabilistic program is given by a sequence of distribution transformations, starting from an initial distribution \( d_0 \). We let \( d_i \sim_{\text{prob}} d_{i+1} \) when \( d_{i+1} \sim_{\text{prob}} T(d_i) \) and let \( \sim_{\text{prob}} \) be the transitive closure of \( \sim_{\text{prob}} \). The probability that a program \( P \) terminates after \( n \) steps starting with initial distribution \( d_0 \) is \( p_n = \sum_{s' \in \mathcal{S}} \text{prob}(s, s') d_n(s) \).

We let \( \text{Pr} \mathbb{P}[P; \varphi] \) be the probability that, starting from a given initial distribution \( d_0 \), command \( P \) completes with a final memory that meets condition \( \varphi \), that is, \( \text{Pr} \mathbb{P}[P; \varphi] = \lim_{n \to \infty} \sum_{s' \in \mathcal{S}} \text{prob}(s, s') d_n(s) \).

(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 \((P, \mu_x)\).

We assume that all algorithms are probabilistic and computable in time bounded by some polynomials in \( \eta \), the security parameter. Intuitively, \( \eta \) represents the lengths of the keys. In order to avoid passing \( \eta \) explicitly, we assume that \( x_\eta \) 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 \( \eta \) (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 \( d_0(\eta) \) where all the weight is given to \((P, \mu_x(x_\eta \ldots \eta))\). We overload \( \text{Pr} \mathbb{P}[P; \varphi] \) to denote the probability function parameterized by \( \eta \).

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 \( \eta \), 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 \( \eta \). We recall the definition of negligible functions:

**Definition 10 (Negligible function).** A function \( f : \mathbb{N} \to \mathbb{R} \) is negligible when, for all \( c \geq 0 \) 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 \( b \) uniformly 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 \( V \) when for all polynomial commands

- \( J \) writing \( V \setminus U : \text{wt}(J) \subseteq V \setminus U \);
- \( B_b \) for \( b = 0,1 \) writing outside \( V : \text{wt}(B_b) \cap (V \cup U) = \emptyset \);
- \( T \) reading \( V \), writing \( g : \text{rv}(T) \subseteq V : g \notin \text{wt}(J, B_0, B_1, \vec{A}) \).
and some variable \( b \not\in v(J, B_0, B_1, T) \) in the command

\[
\text{CNI} = b := \{0, 1\};
J; \text{if } b = 0 \text{ then } B_0 \text{ else } B_1;
\]

\( P \)

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

In the game of the definition, \( J; \text{if } b = 0 \text{ then } B_0 \text{ else } B_1 \) probabilistically initializes 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 \( \langle J; B_0; P, d_b(\eta) \rangle \downarrow d_b(\eta) \) for the same range of commands for \( J \) and \( B_0 \). 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, \( \Gamma \) a policy for its variables, and \( \alpha, \epsilon \in \Gamma \).

\( P \) is computationally non-interferent against \( \alpha \)-adversaries when, for both \( V, U \models V' \), \( 0 \) and \( V, U \models V', V'' \models \epsilon v\nu(P) \), and for all polynomial commands

- \( J \) writing \( V \) to \( U : v\nu(J) \subseteq V \setminus U \);
- \( B_b \) for \( b = 0,1 \) writing outside \( V : v\nu(B_b) \cap V = \emptyset \);
- \( \bar{A} \)-adversaries;
- \( T \) reading \( V, \) writing \( g : rv(T) \subseteq V; g \not\in v\nu(J, B_0, B_1, \bar{A}) \);

and some variable \( b \not\in v(J, B_0, B_1, P, \bar{A}, T) \) in the command

\[
\text{CNI} = b := \{0, 1\};
J; \text{if } b = 0 \text{ then } B_0 \text{ else } B_1;
\]

\( P[\bar{A}] \)

if we have \( |\text{Pr}[\text{CNI}; \bigwedge_{x \in rv(T)} x \not\in \perp] = 1, \text{ then the advantage } |\text{Pr}[\text{CNI}; T; b = g] - \frac{1}{2}| \) is negligible.

The game of this definition performs initialization as in Definition 11, then runs \( P[\bar{A}] \), and finally tests the resulting memory, as in Definition 7. The condition \( \bigwedge_{x \in rv(T)} x \not\in \perp \) 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 \( G_\epsilon, E, \) and \( D \) 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 \( \eta \).

An asymmetric encryption scheme is a triple of algorithms \( (G_\epsilon, E, D) \) such that

- \( G_\epsilon \), used for key generation, generates over publickeys \( \times \) secretkeys;
- \( E \), used for encryption, ranges over ciphertexts;
- \( D \), used for decryption, ranges over plaintexts and is such that, for all \( k_c, k_a \models G_\epsilon(\cdot) \) and \( m \in \text{plaintexts} \), we have \( D(E(m, k_c), k_a) = m \).

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 (at or 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 \( \epsilon \) for membership test.

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

\[
E \triangleq b := 0 \text{ then } m := E(x_0, k_c) \text{ else } m := E(x_1, k_c);
\]

\( \log := \log + m \)

\( D \triangleq \begin{cases} m \in \log \text{ then } x := 0 \text{ else } x := D(m, k_a) \end{cases} \)

\( \text{CCA} \equiv b := \{0,1\}; \log := \text{nil}; k_c, k_a := G_\epsilon(\cdot); A[E, D] \)

The encryption scheme \( (G_\epsilon, E, D) \) provides indistinguishability under adaptive chosen-ciphertext attacks when the advantage \( |\text{Pr}[\text{CCA}; b = g] - \frac{1}{2}| \) is negligible for any polynomial command context \( A \) with \( b, k_a \not\in v\nu(A) \) and \( b, k_a, k_c, \eta, \log \not\in v\nu(A) \).

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 \( G_e, \mathcal{E}, \) and \( D \) 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 \( A = g : = \{0,1\} \) 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 \( x_{1H}, x'_{1H}, x'_{2H} \), and \( k_e \). The command

\[
k_e, k_d := G_e(\cdot); x_{1H} := \mathcal{E}(y_{1H}, k_e); x'_{1H} := \mathcal{E}(0, k_e)
\]

is computationally non-interferent (CNI). In particular, we have \( x_{1H} = x'_{1H} \) with negligible probability, even if \( y_{1H} = 0 \), so any IND-CCA2 encryption function must be probabilistic.

Conversely, none of the three commands

\[
\begin{align*}
P_1 & := x_{1H} := \mathcal{E}(0, y_{1H}) \\
P_2 & := k_e, k_d := G_e(\cdot); x_{1H} := \mathcal{E}(y_{1H}, k_e); x'_{1H} := \mathcal{E}(k_d, k_e) \\
P_3 & := k_e, k_d := G_e(\cdot); k'_e, k'_d := G_e(\cdot); \quad (\text{if } y_{1H} \text{ then } k'_e := k_e) \quad x_{1H} := \mathcal{E}(y_{1H}, k'_e)
\end{align*}
\]

is CNI for some IND-CCA2-secure encryption schemes: in the first case the encryption function is not properly used since \( y_{1H} \) is not a key; in the second case, there is a key cycle \((k_e, k_d)\) 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 \( x_{1H} \) 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 \( x_1 \):

\[
(P_u)_{u=0,1} = \begin{cases} 1 & \text{if } e_u = u \text{ then } h_u := D(x_u, k_d); \\
1 & \text{else } x_{1-u} := \mathcal{E}(s_u, k_e)
\end{cases}
\]

The command \( k_e, k_d := G_e(\cdot) \) while \( e' \) do \((P_2, P_1)\) does not leak any information on \( x_u \), even if the adversary controls the values of \( e', e_u, k_u, x_u \) 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 \( \text{sigkeys}, \text{verifykeys}, \text{signedtexts}, \) and plaintexts be sets of polynomially-bound bitstrings indexed by \( \eta \).

A signature scheme is a triple of algorithms \((G_s, S, V)\) such that

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

For convenience, we assume that \( V \) is deterministic, so that we can use test expressions \( V(e, e', e'' \cdot) \) 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

\[
S := x := S(m, k_s); \log := \log + m; \\
CMA := k_s, k_v := G_s(\cdot); \log := \log + A[S]; \quad \text{if } m \in \log \text{ then } b := 0 \text{ else } b := V(m, x, k_v)
\]

The signature scheme \((G_s, S, V)\) 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 \( k_s, k_v, \log, \eta \).

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 \( k_s, \log, \eta \).

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 \( \eta \), the sets \( \text{plaintexts} \) include all encrypted and signed values.

#### 4.1 Types

We supplement the data type \( \text{Data} \) 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:

\[
\tau ::= \begin{cases} t(\ell) & \text{Security types} \\
\text{Data}: \tau K & \text{Data types for payloads} \\
\text{Enc} \tau K & \text{Data types for encryption} \\
\text{Ks} F \tau K & \text{Data types for signing}
\end{cases}
\]

\[
\begin{align*}
t & ::= [\text{Data}] & Security \text{ types} \\
\text{Enc} \tau K & \text{Data types for encryption} \\
\text{Ks} F \tau K & \text{Data types for signing}
\end{align*}
\]
precisely type expressions of the form $S_{\tau K}(t)$ grouped by their key-generation commands. These labels are at-
static key labels $\ell$ where $t$ and $k$ 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 correspond-
tags usable with the key to the security type of the correspond-
Static key labels The labels $K$ are used to keep track of keys, grouped by their key-generation commands. These labels are at-
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 $S(t \rightarrow m, s)$ where $t$ is a
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 $S(t \rightarrow m, s)$ where $t$ is a

Example 6 (Tagged signatures). The command context

\[
P[\_1] = k_s, k_v, := g_s(); \quad y_{LL} := S(t_0 + x_{IH}, k_s); z_{LL} := x_{IH};
\]
\[
y'_{LL} := S(t_1 + x'_{IH}, k_s); z'_{LL} := x'{IH};
\]
\[
if V(t_0 + z_{LL}, y_{LL}, k_s) then h_{IH} := z_{LL}
\]
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 $t_0$ and $t_1$ is not CNI for integrity, as can be seen for an adversary that overwrites the first value and signature with the second ones:

\[
J = x_{LL} = 0; x'_{IH} := 1
\]
\[
B_0 = z_{LL} := b
\]
\[
A = if z'_{LL} then (y_{LL} := y'_{LL}; z_{LL} := z'_{LL})
\]
\[
T = if x_{LL} = x'_{IH} then g := 0 else g := 1
\]

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 $K \tau K$ 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 $\text{OpVer}$ except for its cryptographic data types:

\[
\begin{align*}
\text{GENE} & \quad \Gamma(k_e) = Ks F K(\ell_e) \quad \Gamma(k_s) = Ks F K(\ell_s) \\
\quad & \quad C(\tau) \leq C(\ell_s) \\
\quad & \quad \vdash k_s, k_v := g_s() : \ell_s \cap \ell_v
\end{align*}
\]

\[
\begin{align*}
\text{ENCRYPT} & \quad \Gamma(x) = \text{Enc} \tau K(\ell_k) \quad I(\tau) \leq I(x) \\
\quad & \quad \vdash x := E(y, k_s) : \ell_x
\end{align*}
\]

\[
\begin{align*}
\text{SIGN} & \quad \Gamma(k_e) = Ks F K(\ell_e) \quad F(t) = \tau \quad \Gamma(x) = \text{Sig} \tau(\ell_s) \\
\quad & \quad \vdash y : \tau \quad I(\tau) \leq \ell_x \quad I(x) \leq I(y) \\
\quad & \quad \vdash x := S(t, y, k_s) : \ell_x
\end{align*}
\]

\[
\begin{align*}
\text{VERIFY} & \quad \Gamma(k_e) = Ks F K(\ell_e) \quad F(t) \leq \Gamma(x) \\
\quad & \quad \vdash y : \ell(t) \\
\quad & \quad \vdash m : \text{Sig} \ell(t)(\ell_x) \\
\quad & \quad \vdash k_s : Ks F K(\ell_s) \\
\quad & \quad \vdash \text{OpVer}(t + y, m, k_s) : Data(\ell_t) \\
\quad & \quad \vdash V(t + y, m, k_s) : Data(\ell_t)
\end{align*}
\]

\[
\begin{align*}
\text{GENS} & \quad \Gamma(k_e) = Ks F K(\ell_e) \quad \Gamma(k_s) = Ks F K(\ell_s) \\
\quad & \quad C(\tau) \leq C(\ell_s) \\
\quad & \quad \vdash k_s, k_v := g_s() : \ell_s \cap \ell_v
\end{align*}
\]

\[
\begin{align*}
\text{PROBFUN} & \quad \Gamma(x) = \text{Data} \ell(t) \quad \text{for } x \in \ell \\
\quad & \quad \vdash \ell(\Gamma(x)) \leq \ell_x \\
\quad & \quad \vdash f() : \ell
\end{align*}
\]

\[
\begin{align*}
\text{OPVER} & \quad \vdash y : \ell(t) \\
\quad & \quad \vdash m : \text{Sig} \ell(t)(\ell_x) \\
\quad & \quad \vdash k_s : Ks F K(\ell_s) \\
\quad & \quad \vdash \text{OpVer}(t + y, m, k_s) : Data(\ell_t) \\
\quad & \quad \vdash V(t + y, m, k_s) : Data(\ell_t)
\end{align*}
\]

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 $\text{OpVer}$ except for its cryptographic data types:

\[
\begin{align*}
\text{GENE} & \quad \Gamma(k_e) = Ks F K(\ell_e) \quad \Gamma(k_s) = Ks F K(\ell_s) \\
\quad & \quad C(\tau) \leq C(\ell_s) \\
\quad & \quad \vdash k_s, k_v := g_s() : \ell_s \cap \ell_v
\end{align*}
\]
have identical data types. The typing of \( P \) records in \( \ell \) 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 (\textsc{assign}), there is no constraint relating the integrity of \( \ell \) 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 \( k_s \) and \( k_d \) with the same key-label and payload type \( \tau \)—the type of plaintexts that can be encrypted and decrypted.

The constraint \( C(\tau) \leq_C C(\ell_d) \) imposes the condition that the 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 \( k_s \), \( y \), and \( x \) involved in encryption; these types are related by \( \tau \) and \( K \), which describe the typing assumptions for encryption with key \( k_s \).

The label \( \ell \) in the typing of \( k_s \) records the flow from \( k_s \) to \( x \) (by subtyping, we have \( I(\ell(k_s)) \leq \ell \)). The hypothesis \( I(\tau) \leq I(x) \) 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 \( \tau \) and \( K \).

The label \( I(x) \) records flows from \( y \) and \( k_d \) to \( x \), as an normal assignment.

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

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

\[
\begin{align*}
k_s, k_d := \mathcal{G}_e();
\lambda' \mathcal{E}(k_s, k_s);
\lambda_d := \mathcal{K}_d();
k_d := \mathcal{K}_d;
\ell := \mathcal{D}(\lambda_d, k_d);
y := \mathcal{E}(s, k)
\end{align*}
\]

For the payload types \( \tau' = \text{Data}(\ell') \) and \( \tau = \mathcal{K} \tau' K'(\ell) \), we define \( I(\tau) \) as follows:

\[
\begin{align*}
\Gamma(k_s) &= \mathcal{K} \tau K(\ell) \\
\Gamma(k'_s) &= \tau \\
\Gamma(\lambda_d) &= \mathcal{K} \tau' K'(\ell) \\
\Gamma(s) &= \tau'
\end{align*}
\]

In case \( \ell \leq L \) and \( \ell' \not\leq L \), this command is insecure because the key \( k_d \) used for decryption does not match the key \( k_s \) 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

\[
\begin{align*}
k_e, k_d := \mathcal{G}_e();
\lambda' \mathcal{E}(k_s, k_s);
\lambda_d := \mathcal{K}_d();
\lambda := \mathcal{D}(\lambda_d, k_d);
y := \mathcal{E}(s, k)
\end{align*}
\]

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 \( y \) uninitialzed. The program is typable using the same \( \Gamma \) of Example 7 extended with types \( \mathcal{S}_\text{Enc} \tau K'K(\ell)(\ell) \) for \( \lambda_d \). \( k \) is \( \mathcal{K} \mathcal{F} K \mathcal{K}(\ell) \) for \( k \). The signing key type \( \mathcal{K} \mathcal{F} K \mathcal{K}(\ell) \) is robust at \( \alpha \) when either \( C(\tau) \leq_C C(\alpha) \) or \( I(\ell) \leq I(\alpha) \).

The signing key type \( \mathcal{K} \mathcal{F} K \mathcal{K}(\ell) \) is robust at \( \alpha \) when either \( I(F(\ell)) \leq I(\alpha) \) for all \( t \in \text{dom}(F) \) or \( C(\ell) \not\leq_C C(\alpha) \).

The security-type policy \( \Gamma \) is robust at \( \alpha \) when all its encryption and signing key types are robust at \( \alpha \).

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 \( \alpha \), 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 17** (Robust policy). Let \( \Gamma \) be a policy and \( \alpha \in \mathcal{L} \).

The encryption key type \( \mathcal{K} \mathcal{E} \tau \mathcal{K}(\ell)K(\ell) \) is robust at \( \alpha \) when either \( C(\tau) \leq_C C(\alpha) \) or \( I(\ell) \leq I(\alpha) \).

The signing key type \( \mathcal{K} \mathcal{F} K \mathcal{K}(\ell) \) is robust at \( \alpha \) when either \( I(F(\ell)) \leq I(\alpha) \) for all \( t \in \text{dom}(F) \) or \( C(\ell) \not\leq_C C(\alpha) \).

The security-type policy \( \Gamma \) is robust at \( \alpha \) when all its encryption and signing key types are robust at \( \alpha \).

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 \( \alpha \), 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 \textsc{check} exclusively assign \( V_{\text{inf}} \) in \( P \); and \( P \) never reads uninitialized variables in \( V_{\text{inf}} \).
2. Each signing-key label/that 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 α ∈ 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 6.)

\[
\text{Init}_s(k_s, k_v) = k_s, k_v := G_s() \]

\[
\text{Init}_t(k_e, k_d) = k_e, k_d := G_t() \]

\[
\text{Read}(x \leftarrow x_e, x_s, x'_e, k_d, k_s, t)[P] = \]

\[
\begin{cases} \text{if } V(t + x_e, x_s, k_s) \text{ then } (x'_e := x_e; x := D(x'_e, k_d); P) \\
\end{cases} \]

\[
\text{Write}(x_e, x_s \leftarrow x'_e, k_e, k_d, t) = \\
\]

\[
(x'_e := E(x, k_e); x_s := S(t + x'_e, k_s); x := x'_e) \]

Command \text{Init}_s generates keys for signing and verification. Command \text{Init}_t generates keys for encryption and decryption. Command context \text{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 \text{Init}_p runs. Otherwise the command silently fails. Conversely, command \text{Write} writes 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 \( \alpha \), 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

\[
\text{Init}_s(H, k_s, k_v); \text{Init}_t(H, k_e, k_d); \]

\[
\text{Write}(x_e, x_s \leftarrow H, k_e, k_d, k_v) \]

\[
\text{Read}(H, k_e, k_d, k_v)[P] \]

The first command line models our hypothesis that \( H \) and \( H' \) share correctly-generated long-term keys \((k_e^H, k_d^H, k_v^H)\), with a private signing key for \( H \) \((k_s^H)\) and a private decryption key for \( H' \) \((k_s^{H'})\). The third line represents session-key generation by \( H \); four local keys are generated. These keys are then concatenated (in \( H, k_e \) 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 \( H, k_e, H, k_s, H, k_v, \) 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 \( \bar{h} \) 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

\[
S := (P)^+ \mid S; S
\]

where each thread \((P)^+\) is annotated with a unique thread identifier \( i \). Let \( \gamma \) 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 \( x' \) 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 \( \Gamma \), 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 \( k_h^s \) and knows every other host’s verification key \( k_h^v \). We initialize these keys by running the command

\[
\text{Init} \bar{h} = (k_h^s, k_h^v) := G_i(); \forall h \in \bar{h}
\]

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 \( i.x_s \) are local
buffers for the encrypted value of \( x \); \( i.x \) 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

\[
\text{Init}_X = (k_e(x), k_d(x) : = \emptyset(1)); x \in X
\]

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

### Translating security policies

We now translate \( \Gamma \), thereby giving types to all target variables: we let \([\Gamma]\) be the security policy that coincides with \( \Gamma \) on \( \text{dom}(\Gamma) \setminus X \) 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 \( \ell' \) be a security level such that \( I(\ell') = I(\ell') \) and \( C(\ell') = C(\ell') \). (Alternatively, we could parameterize the translation with target security labels, and write a set of constraints to be satisfied in order to preserve typeability.)

For every host \( h \in \mathcal{H} \), we let

\[
[\Gamma](k_e^h) = Ks \ F Ks(k_e^h) \quad [\Gamma](k_d^h) = Kv \ F Ks(k_d^h)
\]

where \( Ks \) is a unique static key label (no other host will have the same label), \( F \) is a mapping from tags to security types, and \( k_e^h, k_d^h \) 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 \( \tau = \text{Data}(\ell) \) for some \( \ell \). Let \( \ell_e \in \mathcal{L} \) such that \( I(\ell_e) = I(\ell) \) and \( C(\ell_e) = C(\ell) \).

We let

\[
[\Gamma](x_e) = \text{Enc} \ \tau \ Ks(k_e^h) \quad [\Gamma](x_s) = \text{Sig} \ \tau \ Ks(k_e^h) \quad [\Gamma](x_e) = \text{Ke} \ \tau \ Ks(k_e^h) \quad [\Gamma](x_d) = \text{Kd} \ \tau \ Ks(k_e^h) | C(\ell) = C(\ell_e) \wedge I(\ell_e) \leq I(\ell)
\]

where \( Ks \) 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

\[
[\Gamma](i.x) = \tau \quad [\Gamma](i.x_e) = \text{Enc} \ \tau \ Ks(k_e^h) \quad [\Gamma](i.x_s) = \text{Enc} \ \tau \ Ks(k_e^h)
\]

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

\[
\ell_e^i \leq \ell_e \quad I(\ell_e^i) \leq I(\ell_e) \quad F(\cdot' + i) = \text{Enc} \ \tau \ Ks(k_e^h)
\]

### 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 \( \{P_i\} \) by first decoring 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 \( \{P_i\} \) of \( S \), we let \( R'[\_] \) be the composition of command contexts

\[
\text{Read}(i.x_d \leftarrow x_e, x_s, i.x_e, k_e(x), k_d(x), k_e(w), \cdot' + w)[\_]
\]

for all annotated variables \( x \in w \in \{P_i\} \cap X \) with \( w \neq i \), let \( P_i' \) be the sequence of assignments \( i : = i.e. \) for the variables \( i.x_e \) assigned in \( R' \), and let \( W_1 \) be the sequential composition of commands

\[
\text{Write}(x, x_e \leftarrow i.x, i.x_e', k_e(x), k_d(x), k_e(i.x) + i, \cdot' + i)
\]

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

\[
\{P_i\} = R'[P_i' ; P_i(i.x/x \ for \ x \in X) ; W^i]
\]

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

\[
(y_{hh} : = 0); (if y_{hh} = 0 then x_{hh} : = 1)^3
\]

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:

\[
P_0 = 0.y_{hh} : = 0; 0.y_{hh} : = \text{Enc}(0.y_{hh}, k_{e}) \quad y_{hh} : = S(y_{hh} + 0 + 0.y_{hh}, k_{s}); y_{ll} : = 0.y_{hh}
\]

\[
P_1 = \text{if } y_{hh} : = 0 + 0.y_{hh}, y_{ll} : = k_{d} \quad then(1.y_{hh}, e : = y_{ll}, 1.y_{hh} : = D(1.y_{hh}, k_{d}); 1.y_{hh} : = 1.y_{hh}, e, if 1.y_{hh} = 0 then x_{hh} : = 1)
\]

### 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 protocols 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 \( (P, \mu) \) such that \( \mu \) is uninitalized on \( X \), we let \( \mu_d \) be the memory defined as follows: for every host \( h \in \mathcal{H} \), and for every \( x \in X \) and thread \( i \) of \( P \) such that \( x \) occurs in \( i, k_e^h, k_d^h, x_e, x, k_e(x), k_d(x), i.x, i.x_e, i.x_e' \) and \( i.x \) are defined and uninitialized; for every \( x \in \text{dom}(\mu) \setminus X \), \( \mu_d(x) = \mu(x) \); and we let \( [[(P, \mu)]] \) be the distribution of configurations such that \( (\mu, \mu) \).
THEOREM 5 (Functional Adequacy). Let $S$ be a correctly-annotated source system and $\mu$ a memory such that $(S, \mu) \leadsto (\sqrt{\cdot}, \mu')$.

We have $[[S[\text{skip}], \mu]] \leadsto \sum_d d_{\mu}$ for a distribution $d_{\mu}$ that gives probability 1 to configurations $(\sqrt{\cdot}, \mu'_{\gamma})$ such that $\mu'_{\gamma}$ coincides with $\mu'$ on their joint domain.

In the theorem, the distribution of final states $d_{\mu}$ may give positive probabilities to a range of different memories $\mu'_{\gamma}$, 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 $\mu$ is fixed, we do not need to assume that $P$ is polynomial: for a given run, we can select sets plaintext 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 $\Gamma \vdash \alpha$, then $[[\Gamma]] \vdash [[S]]$.

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

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

If $[[\Gamma]]$ is a robust policy at $\alpha$, then $[[S]]$ is computationally non-interferent against $\alpha$-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


