A Security-Preserving Compiler for Distributed Programs

From Information-Flow Policies to Cryptographic Mechanisms

Cédric Fournet
Microsoft Research
Cambridge
United Kingdom
fournet@microsoft.com

Gurvan Le Guernic
MSR–INRIA Joint Centre
Orsay
France
GLGuern@gmail.com

Tamara Rezk
INRIA Sophia Antipolis
Méditerranée
France
Tamara.Rezk@inria.fr

ABSTRACT

We enforce information flow policies in programs that run at multiple locations, with diverse levels of security.

We build a compiler from a small imperative language with locality and security annotations down to distributed code linked to concrete cryptographic libraries. Our compiler splits source programs into local threads; inserts checks on auxiliary variables to enforce the source control flow; implements shared distributed variables using instead a series of local replicas with explicit updates; and finally selects cryptographic mechanisms for securing the communication of updates between locations.

We establish computational soundness for our compiler: under standard assumptions on cryptographic primitives, all confidentiality and integrity properties of the source program also hold with its distributed code, despite the presence of active adversaries that control all communications and some of the program locations. We also present performance results for the code obtained by compiling sample programs.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: Protection Mechanisms

General Terms

Security, Design, Languages

1. INTRODUCTION

The security of distributed systems usually entails the implementation of protection mechanisms based on cryptography to ensure the confidentiality and integrity of information. This involves expert knowledge, as well as attention to many implementation details. Our goal is to let developers focus on high-level security policies and properties of their programs, and use a compiler to generate lower-level protection mechanisms that ensure that the distributed implementation is at least as secure as the source program.

We take information flow security as our specification of security (for an abstract memory model) and also as our model for cryptography in the implementation. In information flow security, policies are expressed by annotating variables with labels from a lattice [see e.g. Myers and Liskov, 2000]. Correct information flow in a program means that an adversary with restricted access to program variables can neither affect the program behavior nor gain knowledge above its security level by interacting with the system. Since this notion of security depends on the semantics of programs, an essential difficulty is to preserve security properties as programs get compiled to concrete implementations. In a distributed implementation, for instance, a network adversary may observe messages sent between hosts, and it may control their scheduling. Inasmuch as these side channels are not apparent in source programs, they must be carefully addressed in the compilation process.

We enforce information flow policies in programs that run at multiple locations, with diverse levels of security. This involves cryptographic protection whenever relatively secure locations (e.g. a client and a server) interact via less secure locations (e.g. an open network).

- In source programs, security depends on a global program semantics, with abstract policies for reading and writing shared memory. These policies enable a simple review of confidentiality and integrity properties.
- In their distributed implementations, shared memory is unprotected, the adversary controls the scheduling, and security depends instead on cryptographic protection.

Our compiler is structured into four stages: slicing, control flow, replication, and cryptography. The first stage slices sequential code with locality annotations into a series of local programs, each meant to run at a single location. After slicing, the second stage protects the control flow of the source program against a malicious scheduler, by generating code that maintains auxiliary variables to keep track of the program state, based on its integrity policy. The replication stage transforms a distributed program (still relying on a global, shared, protected memory) into a program where variables are implemented as local replicas at each location, with explicit updates between replicas. Finally, the cryptography stage inserts cryptographic operations to protect these variable updates, and it generates an initial protocol for distributing their keys.

Our target notions of security are expressed in a computational model of cryptography. In this model, adversaries are probabilistic programs that operate on bitstrings and have limited computational power. This leads us to reason with polynomial-time hypotheses and probabilistic semantics. We could have used instead a symbolic model of cryptography, where adversaries may perform arbitrary computations on abstract algebraic terms (not bitstrings). However, this simpler model would have hidden many cryptographic side channels that are relevant in distributed implementations and problematic for information security. The relation between symbolic and computational models is the subject of active research [see e.g. Abadi and Rogaway, 2002, Backes et al., 2003, Comon-Lundh and Cortier, 2008, Laud, 2008] but it is unlikely that they can be reconciled at the level of details handled by our compiler. Thus, we seek computational soundness directly for information-flow security, rather than for symbolic cryptography.

In prior work, Fournet and Rezk [2008] design a computationally sound type system for cryptography and give a typed translation from non-interferent source programs to their cryptographic implementations. Our theory extends theirs in several directions:

Active Adversaries: Our compiled code is secure against adversaries that control the scheduling between hosts. This reflects a realistic attacker model for distributed programs, where the opponent controls parts of the program (representing for instance the corrupted parties of a protocol) and also controls all interactions between the remaining "honest" parts of the program (representing for instance an open network). In their work, they restrict the control flow of programs and assume that the compiled code follows the source control flow.

Information release: Our safety conditions on source programs are less restrictive and do not require noninterference. Hence, our compiler accepts programs that selectively leak information, and our theorems state that, for all safe source programs, if an adversary can successfully attack our compiled code, then there is also an adversary that can successfully attack the source program.

Efficient use of cryptography: They formalize only asymmetric cryptography. In contrast, we use asymmetric cryptography only for initial key distribution, then rely on symmetric cryptography, which is much more efficient. We also allocate fewer keys and perform simple cryptographic optimizations.

Main Contributions

- We design and implement a compiler from sequential programs with shared memory to distributed programs at least as secure as the source. Our tool combines both symmetric and asymmetric cryptography and yields efficient code.
- We account for a realistic class of active adversaries, which control some components of the system (including the network) and schedule the others.
- We obtain computational soundness theorems for all information flows, both for secrecy and for integrity. (We also show functional correctness, but only for an adversary that implements a reliable network.)
- We report experimental performance results obtained for a series of sample distributed programs.

Related Work Due to lack of space, we discuss only closely related work. We refer to Sabelfeld and Myers [2003] for a survey of information flow security, and to Fournet and Rezk [2008] for a more complete account of cryptographic information flows.

Computational noninterference: Laud [2001] pioneers work on information flow relying on concrete cryptographic assumptions. He introduces computational correctness for encryption in a model with passive adversaries. Our notions of noninterference generalize this property to the active case, and also cover integrity properties.

Secure program partitioning: Jif/Split [Zdancewic et al., 2002, Zheng et al., 2003] is a compiler from information flow typed sequential Java programs to distributed systems with mutual distrust between hosts. Their distributed implementation relies on secure communications, modelled as private channels. We lift this assumption, implement communications using cryptographic mechanisms, and prove them correct under standard cryptographic hypotheses. Hence, our compiler can be seen as a cryptographic back-end for Jif/Split. Unlike Jif/Split, we do not consider code replication but only data replication.

Robustness: A system is robust when an adversary cannot affect the security of information flow [Zdancewic and Myers, 2001, Myers et al., 2006]. Decentralized robustness generalizes this notion to configurations with mutual distrust between principals [Chong and Myers, 2006]. In this work, we rely on similar robustness conditions on source programs.

Contents Section 2 defines our source and target languages. Section 3 defines information flow policies and security properties. Sections 4, 5, 6, and 7 describe the slicing, control-flow, replication, and cryptographic stages of the compiler. Section 8 reports experimental results. Section 9 concludes. Additional definitions, examples, and proofs appear online at `http://www.msr-inria.inria.fr/projects/sec/cflow`.

2. LANGUAGES

In this section, we present a core probabilistic imperative language and its extension to express distribution. We also define concrete distributed programs with explicit scheduling.

Core Language We use a while-language based on transparent shared memory, with the following grammar:

$$\begin{array}{lll} e & ::= & x \mid op(e_1, ..., e_n) \\ S, P, A & ::= & x := e \mid x := f(x_1, \ldots, x_n) \mid \textbf{skip} \mid S ; S \\ & \mid & \textbf{if}\ e\ \textbf{then}\ S\ \textbf{else}\ S \mid \textbf{while}\ e\ \textbf{do}\ S \\ & \mid & x := \textbf{declassify}(e, \ell) \end{array}$$

where op and f range over deterministic and probabilistic n-ary functions, respectively, with $n \geq 0$. Expressions e consist of variables and operations, including standard boolean and arithmetic constants and operators. Programs and commands S consist of variable assignments, using deterministic expressions and probabilistic functions, composed into sequences, tests, and loops. (The assignment $x := \textbf{declassify}(e, \ell)$ behaves as $x := e$; it is explained in Section 3.) We use curly brackets $\{S\}$ for parenthesizing commands. We let $\mathrm{wv}(S)$ be the set of variables written by S, let $\mathrm{rv}(S)$ be the set of variables read by S, and let $\mathrm{v}(S)$ be $\mathrm{wv}(S) \cup \mathrm{rv}(S)$.

Although our language does not feature procedure calls, we can use command contexts to range over programs with access to fixed, privileged procedures (sometimes called "oracles" in cryptography) using shared variables for passing their input and output parameters. An n-ary command context, written $P[\__0, \ldots, \__{n-1}]$, is a term obtained from the grammar of commands extended with placeholders for commands $\__i$ (and, more generally, for command contexts $\__i[P_1, \ldots P_{k_i}]$). For instance, $P_0; \_[P']$ represents a command, parameterized by a command context $A[\_]$, that first runs P_0 then runs A, which may in turn call P' any number of times.

Probabilistic Semantics The semantics of each probabilistic function is given by a discrete parametric probability distribution. We write $\{0, 1\}$ for the fair "coin-tossing" function that returns either 0 or 1 with probability $\frac{1}{2}$. We use probabilistic functions mainly to model cryptographic algorithms as commands.

Program configurations are of the form $\langle P, \mu \rangle$ where P is a program and μ is a *memory*, that is, a function from variables to values. The special program $\surd$ represents termination. The operational semantics of commands is given as Markov chains between program configurations, with probabilistic steps $s \rightsquigarrow_p s'$ induced by the probabilistic functions (see the full paper). We lift these reduction steps to configuration distributions, and write $d \rightsquigarrow d'$ when, for all configurations s', $d'(s') = \sum_{s \rightsquigarrow_p s'} p \times d(s)$. We write $\rightsquigarrow^*$ for the transitive closure of $\rightsquigarrow$. We define the semantics of a program P with initial memory μ as follows: d_0 is the configuration distribution such that $d_0(\langle P, \mu \rangle) = 1$; $d_i \rightsquigarrow d_{i+1}$ for $i \geq 0$; and $\Pr[P; \varphi]$ is the probability that P completes with a final memory that meets condition φ: $\Pr[P; \varphi] = \lim_{n \to \infty} \sum_{s = \langle \surd, \mu \rangle | \varphi} d_n(s)$. (The limit exists because the sum increases with n and is bounded by 1.)

Source Language, with Locations Let $a, b, \ldots \in \mathcal{H}$ be a finite set of hosts, intended to represent units of trust (principals) and of locality (runtime environments). We extend the grammar of the core language (S) with locality annotations:

$$S^+, P^+ \quad ::= \quad \ldots \mid a \colon S^+$$

The locality command $a \colon S^+$ states that command S^+ should run at host a. This programming abstraction hides the implementation details for transferring control between the current host and a before and after running command S^+. Locality commands can be nested, as in $a \colon \{P_0; b \colon P_1; c \colon P_2\}$. We assume that every source program has a locality command at top level, setting an initial host.

Since memory is transparently shared between hosts, locality annotations do not affect our command semantics.

Target Language, with Explicit Scheduling A distributed program is just a series of commands in the core language, each command intuitively running at a single host. We refer to these commands as threads. (Pragmatically, our compiler groups threads running at the same host into a single host command that locally schedules its threads.)

To model the intended behavior of a distributed program with n threads, in particular to state its correctness in the absence of an adversary, we define an n-ary command context N_n that implements a round-robin scheduler. This command context uses a global program counter variable $next$ that indicates which thread should run next, with a special value $\mathtt{stop}$ to indicate the end of the execution.

Definition 1 (n-ary Scheduler).

$$N_n[\_1, \ldots, \_n] \doteq \textbf{while } next \neq \mathtt{stop} \textbf{ do } \{\_1; \ldots; \_n\}$$

This context may represent a public network, for instance, with communications between hosts using messages in shared memory.

Finally, we model a compiler $\mathcal{C}$ as a function from source programs P^+ to series of commands $Q_0, Q_1, \ldots Q_n = \mathcal{C}(P^+)$ where Q_0 is a distinguished initialization command and $Q_{i>0}$ are commands representing threads, meant to be executed as

$$next := \mathtt{start}; Q_0; N_n[Q_1, \ldots, Q_n]$$

To study the security properties of this compiled program, we will replace N_n with some unknown command context A representing an active adversary that controls the scheduler.

3. COMPUTATIONAL NONINTERFERENCE

We briefly recall standard notions of information flow policies, then define our main security properties.

Security Labels We annotate every variable with a security label. These labels specify the programmer's security intent, but they do not affect the operational semantics.

The security labels form a lattice $(\mathcal{L}, \leq)$ obtained as the product of two lattices, for confidentiality levels $(\mathcal{L}_C, \leq_C)$ and for integrity levels $(\mathcal{L}_I, \leq_I)$. We write $\bot_\mathcal{L}$ and $\top_\mathcal{L}$ for the smallest and largest elements of $\mathcal{L}$, and $\sqcup$ and $\sqcap$ for the least upper bound and greatest lower bound of two elements of $\mathcal{L}$, respectively. We write $\bot_C$, $\bot_I$, $\top_C$, $\top_I$ for the smallest and largest elements of $\mathcal{L}_C$ and $\mathcal{L}_I$, respectively.

For a given label $\ell = (\ell_C, \ell_I)$ of $\mathcal{L}$, the confidentiality label ℓ_C specifies a read level for variables, while the integrity label ℓ_I specifies a write level; the meaning of $\ell \leq \ell'$ is that ℓ' is more confidential (can be read by fewer entities) and less trusted (can be written by more entities) than ℓ [Myers et al., 2006]. 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 $I(\ell) \leq_I I(\ell')$.

Memory and Host Policies We represent our memory policy as a function Γ from variables to security labels. For brevity, we sometimes write $I(x)$ (resp. $C(x)$) instead of $I(\Gamma(x))$ (resp. $C(\Gamma(x))$).

We extend Γ to represent host policies as a map from host names to security labels. Host policies are used to establish a control flow protocol (see Section 5) and to select cryptographic protection. Our intent is that host a may read the variables V_a^C and write the variables V_a^I, defined as

$$V_a^C \doteq \{x \mid C(x) \leq_C C(a)\} \qquad V_a^I \doteq \{x \mid I(a) \leq_I I(x)\}$$

Adversaries Our security properties are parameterized by the power of the adversary, defined as a pair $\mathcal{A} = (\mathcal{A}_C, \mathcal{A}_I)$ of subsets of the security lattices:

- $\mathcal{A}_C \subset \mathcal{L}_C$, the *public labels*, is a non-empty downward-closed subset of the confidentiality lattice;
- $\mathcal{A}_I \subset \mathcal{L}_I$, the *tainted labels*, is a non-empty upward-closed subset of the integrity lattice.

In the rest of the paper, we often assume a fixed policy and adversary $\mathcal{A}$, and let

$$V_\mathcal{A}^C = \{x \mid C(x) \in \mathcal{A}_C\} \qquad \overline{V}_\mathcal{A}^I = \{x \mid I(x) \notin \mathcal{A}_I\}$$

An *active adversary command*, ranged over by A, is a core command that reads only public variables ($\mathrm{rv}(A) \subseteq V_\mathcal{A}^C$) and writes only tainted variables ($\mathrm{wv}(A) \cap \overline{V}_\mathcal{A}^I = \emptyset$). In particular, A can always read variables with confidentiality label $\bot_C$ and always write variables with integrity label $\top_I$.

For example, let $\mathcal{L}_4$ be the 4-point security lattice defined by the product of the confidentiality lattice $\{L \leq_C H\}$ and the integrity lattice $\{H \leq_I L\}$. In this lattice, the adversary $\mathcal{A} = (\{L\}, \{L\})$ yields adversary commands that can read low-confidentiality variables and write low-integrity variables. For brevity, elements of $\mathcal{L}_4$ are written *HL*, *HH*, *LL*, and *LH* in the rest of the paper.

In the general case, one can define the adversary by indicating the subset of hosts that may have been compromised and letting $\mathcal{A}_C$ and $\mathcal{A}_I$ be the closures of their confidentiality and integrity labels. This ensures that any commands at these hosts become valid adversary commands.

Our implementation depends both on the security policy and on the structure of the lattices, but it does not depend on the choice of a particular adversary.

Indistinguishability Games In computational models of cryptography, security properties are often expressed as games, coded as commands that sample a secret boolean $b := \{0, 1\}$ then implement a protocol that interacts with an adversary, also modelled as commands. The goal of the adversary is to write into some variable g its guess as to the value of b: the adversary wins when $b =_0 g$ (where the operator $=_0$ is boolean equality, true iff both or none of its operands equal 0). The trivial adversary $g := \{0, 1\}$ wins with probability $\frac{1}{2}$, so we are interested in a bound on the *advantage* of an adversary, defined as the probability that $b =_0 g$ minus $\frac{1}{2}$. The protocol is deemed secure when, for a game that involves only commands that run in polynomial time, this advantage is negligible in the security parameter (usually the length of the keys used in the protocol).

Confidentiality and Integrity We define security properties for probabilistic command contexts as computational variants of noninterference, expressed as games.

For confidentiality, our property is parameterized by active adversaries plus three commands that initialize variables that the adversary can read (set by J) or not (set by either B_0 or B_1).

DEFINITION 2 (COMPUTATIONAL CONFIDENTIALITY). *Let Γ be a policy, $\mathcal{A}$ an adversary, and P a command context. Let J, B_0, and B_1 range over polynomial commands such that $\mathrm{wv}(B_b) \cap V^C_{\mathcal{A}} = \emptyset$ for $b = 0, 1$. Let $\vec{A}$ range over tuples of adversary contexts such that $P[\vec{A}]$ is a polynomial command. Consider the command*

$$\mathit{CNI}_C \doteq b := \{0,1\}; J; \textbf{if } b = 0 \textbf{ then } B_0 \textbf{ else } B_1; P[\vec{A}]$$

where $g \notin \mathrm{v}(J, B_0, B_1, P)$ and $b \notin \mathrm{v}(J, B_0, B_1, P, \vec{A})$.

P is computationally confidential (CNI_C) for J, B_0, B_1, and $\vec{A}$ when $|\Pr[\mathit{CNI}_C; b =_0 g] - \frac{1}{2}|$ is negligible.

P is CNI_C for J, B_0, B_1 when this holds for all $\vec{A}$.

P is CNI_C when this holds for all J, B_0, B_1, and $\vec{A}$.

In the definition, the command contexts $\vec{A}$ represent the code of an active adversary that interacts with P and tries to infer the value of b. The adversary "knows" P, J, B_0, and B_1, inasmuch as the definition of $\vec{A}$ may depend on them. Implicitly, the last adversarial piece of code in $P[\vec{A}]$ is supposed to write into g its guess for b.

By definition of the command CNI_C, the value of b affects the initial state of the memory when $P[\vec{A}]$ runs, by running either B_0 or B_1, but only for high-confidentiality variables, which $\vec{A}$ cannot directly read, so $\vec{A}$ can win the game with some advantage only if P somehow leaks information from high-confidentiality to low-confidentiality variables.

The three statements at the end of the definition differ only in their generality. The first statement is for two specific distributions of initial memories, set by $J; B_0$ and $J; B_1$ respectively, and for a specific adversary. The second statement expresses that no such adversary may effectively distinguish between these two specific distributions of initial memories; it may be used to characterize the security of commands that leak some confidentiality information. The third statement expresses that this holds for any such distributions of initial memories, and is a computational variant of noninterference (CNI_C). If we omit the computational hypotheses, run $P[\vec{A}]$ with arbitrary initial low-equivalent memory distributions ρ_b rather than those initialized by $J; B_b$, and compare (exactly) the distributions of low-confidentiality variables after running $P[\vec{A}]$ rather than just the values of g, we retrieve a formulation of probabilistic noninterference.

Let us consider special cases for P and $\vec{A}$:

- If the command context P is of the form $Q_0; \_$ with a single adversary command that runs after Q_0, then computational confidentiality reduces to a notion of noninterference against passive adversaries that observe low-confidentiality memory only after Q_0 completes, but do not interact with Q_0.
- If P is of the form $Q_0; \_; Q_1; \_; \ldots; Q_n; \_$, the adversary $\vec{A}$ consists of a tuple of $n + 1$ commands that run between each of the commands Q_i and represents an active adversary whose execution is interleaved with that of P, but which cannot change the order in which the commands Q_i run.
- If P is of the form $Q_0; \_[Q_1, \ldots, Q_n]$, then the adversary $\vec{A}$ consists of a single n-ary command context that represents an untrusted network or scheduler that can run the commands Q_i any number of times, in any order. This adversary is strictly more powerful than the one above.

We use an (almost dual) security definition for integrity. The definition uses an auxiliary polynomial command T that reads high-integrity variables and writes g after running the interactive computation between our command and the active adversary.

DEFINITION 3 (COMPUTATIONAL INTEGRITY). *Let Γ be a policy, $\mathcal{A}$ an adversary, and P a command context.*

Let J, B_0, B_1, T range over polynomial commands such that $\mathrm{wv}(J) \subseteq \overline{V^I_{\mathcal{A}}} \setminus \mathrm{wv}(P)$ and $\mathrm{wv}(B_b) \cap \overline{V^I_{\mathcal{A}}} = \emptyset$ and $\mathrm{rv}(T) \subseteq \overline{V^I_{\mathcal{A}}}$. Let $\vec{A}$ range over adversary contexts such that $P[\vec{A}]$ is a polynomial command. Consider the command

$$\mathit{CNI}_I \doteq b := \{0,1\}; J; \textbf{if } b = 0 \textbf{ then } B_0 \textbf{ else } B_1; P[\vec{A}]$$

where $b \notin \mathrm{v}(J, B_0, B_1, P, \vec{A}, T)$ and $g \notin \mathrm{v}(J, B_0, B_1, P, \vec{A})$. A run of CNI_I is valid *when every variable x in $\mathrm{wv}(\mathit{CNI}_I) \cap \mathrm{rv}(T)$ is written exactly once.*

P is computationally integral (CNI_I) for J, B_0, B_1, $\vec{A}$, and T when $\Pr[\mathit{CNI}_I \text{ valid}] = 1$ implies that $|\Pr[\mathit{CNI}_I; T; b =_0 g] - \frac{1}{2}|$ is negligible.

P is CNI_I for J, B_0, B_1 when this holds for all $\vec{A}$, and T.

P is CNI_I when this holds for all J, B_0, B_1, $\vec{A}$, and T.

In the definition, the initialization command J sets variables that are not writable by $\vec{A}$ but that are readable by T. The game consists of letting $\vec{A}$ interact with the system and try to force the program behavior to depend on the low-integrity bit b, thus yielding different value assignments to high integrity variables.

Since the adversary $\vec{A}$ may prevent the execution of certain high-integrity threads, the definition imposes an additional condition that excludes this means of communication with T: the variables written by CNI_I and read by T must have been written exactly once. This condition enables us to consider as secure programs that perform checks on low-integrity variables (for instance a signature verification), then assign high-integrity variables only if the checks succeed. Hence, our definition let $\vec{A}$ and T range over commands such that $\vec{A}$ makes a check fail, or T reads those high-integrity variables, but not in the same game. For example, the command context $\_[x := 0]$ is CNI_I, but $\_[x := 0, x := 1]$ is not. As a drawback, the definition accounts for the integrity only of the first assignment to each variable, but this weakness can be mitigated by rewriting P in single-assignment style, as explained in Section 6. (See also the discussion of weak integrity and runtime failures in Fournet and Rezk 2008.)

In the following, we are interested in both confidentiality and integrity, and we say that a command context is *computationally non-interferent* (CNI) when it is both CNI_C and CNI_I.

Source Program Safety Our compilation process makes assumptions on source programs, which we define next. (Compilation may still fail on some inputs, as explained in Section 6; we ruled out safety conditions that would guarantee that the compilation always succeeds as unduly restrictive.)

Figure 1 presents a type system that captures our safety hypotheses on programs with localities. The typing judgments for source

$$\textsc{TSubC}\quad \frac{\vdash P^+ : \ell \qquad \ell' \leq \ell}{\vdash P^+ : \ell'} \qquad\qquad \textsc{TAssign}\quad \frac{\vdash e : \Gamma(x)}{\vdash x := e : \Gamma(x)}$$

$$\textsc{TDeclassify}\quad \frac{\vdash e : \ell \qquad I(\ell) = I(x)}{\vdash x := \mathsf{declassify}(e, \ell) : \Gamma(x)} \qquad\qquad \textsc{TFun}\quad \frac{\vdash \vec{y} : \Gamma(x)}{\vdash x := f(\vec{y}) : \Gamma(x)}$$

$$\textsc{TSeq}\quad \frac{\vdash P_0^+ : \ell \qquad \vdash P_1^+ : \ell}{\vdash P_0^+ ; P_1^+ : \ell} \qquad\qquad \textsc{TCond Local}\quad \frac{\vdash e : \ell \qquad \vdash P_1 : \ell \qquad \vdash P_0 : \ell}{\vdash \textbf{if } e \textbf{ then } P_1 \textbf{ else } P_0 : \ell}$$

$$\textsc{TCond}\quad \frac{\vdash e : \ell \qquad \vdash P_1^+ : \ell \qquad \vdash P_0^+ : \ell \qquad C(\ell) = \bot_C}{\vdash \textbf{if } e \textbf{ then } P_1^+ \textbf{ else } P_0^+ : \ell}$$

$$\textsc{TWhile Local}\quad \frac{\vdash e : \ell \qquad \vdash P_1 : \ell}{\vdash \textbf{while } e \textbf{ do } P_1 : \ell} \qquad\qquad \textsc{TWhile}\quad \frac{\vdash e : \ell \qquad \vdash P_1^+ : \ell \qquad C(\ell) = \bot_C}{\vdash \textbf{while } e \textbf{ do } P_1^+ : \ell}$$

$$\textsc{TLocality}\quad \frac{\vdash P^+ : \ell \qquad I(b) \leq_I I(\ell) \qquad rv(P) \subseteq V_b^C}{\vdash (b\!: P^+) : \ell} \qquad\qquad \textsc{TSkip}\quad \vdash \textbf{skip} : \top^{\mathcal{L}}$$

Figure 1: Typing rules (for a given policy Γ).

commands are of the form $\vdash P : \ell$ where ℓ is a security label. We omit the standard rules for typing expressions, such that $\vdash e : \ell$ when $\Gamma(x) \leq \ell$ for each variable x read in e. This type system is similar but more permissive than those typically used for noninterference [see e.g. Sabelfeld and Myers, 2003] as it accepts programs with explicit declassifications. We discuss some specific rules:

TDECLASSIFY does not prevent explicit confidentiality flows, but enforces that the command be typed with the integrity level of the label that appears in the declassify annotation. Those labels will be subject to robustness conditions.

TCOND LOCAL and TWHILE LOCAL are adapted from standard rules to prevent implicit flows. Their guarded code (P_1 and P_0) range over core commands, which do not contain occurrences of localities. Otherwise, distributed execution may leak information about the guard e: for instance, in the program **if** s_{HH} **then** a : **skip else** b : **skip**, the guarded commands are not local, hence it would be hard to hide whether a or b executes next. Conversely, TCOND and TWHILE allow guarded code with locality commands, but only when the guard e is public.

TLOCALITY excludes trusted code in untrusted localities. This is needed to prevent an adversary that controls the scheduling to trigger an execution of P^+ at b that is not enabled by the control flow of the source program.

DEFINITION 4 (SOURCE PROGRAM SAFETY). *A label $\ell \in \mathcal{L}$ is* robust against *$\mathcal{A}$ when $I(\ell) \in \mathcal{A}_I$ implies $C(\ell) \in \mathcal{A}_C$.*

A source program P^+ is safe for Γ *and $\mathcal{A}$ when P^+ is typable, polynomial, and all its declassification labels are robust against $\mathcal{A}$.*

Typability implies that, in a source program, no untrusted host is allowed to modify trusted variables, neither by modifying them directly, nor by calling a trusted host that modifies high integrity variables. That is, for every locality $a : S^+$ within P^+, we must have $rv(S^+) \subseteq V_a^C$ and $\mathrm{wv}(S^+) \subseteq V_a^I$.

In the absence of declassification, typability guarantees *CNI* [see e.g. Fournet and Rezk, 2008]. Otherwise, the robustness condition ensures that, if the adversary can influence a declassification, then it can also directly access the declassified information [Zdancewic and Myers, 2001]. Thus, depending on the lattice, a program with some declassification (hence not necessarily *CNI*) may still be compiled, with an implementation that provides the same security guarantees as for the source program against the restricted class of adversaries that make it safe [see also Chong and Myers, 2006].

We illustrate each stage of our compilation process with the sample program listed below. Section 8 discusses other examples.

EXAMPLE 1. *For the 4-points lattice $\mathcal{L}_4$ (with two levels of confidentiality and integrity), let S^+ be the source program*

```
a:{
 xHL := 1; yLH := 2;
 while yLH < 3 do {
  yLH := yLH + 4;
  b:{
   if (yLH mod 2) = 1
   then {xHL := xHL + 9}
   else {skip}
  };
  c:{zLH := 5} } }
```

and Γ a policy such that $\Gamma(a) = \Gamma(b) = \Gamma(c) =$ HH for hosts, and $\Gamma(x) =$ HL, $\Gamma(y) = \Gamma(z) =$ LH for variables. S^+ is typable, polynomial, and has no declassify, so it is safe and CNI.

Correctness and Security for Compiled Code We are now ready to specify the intended properties of our compiler, beginning with functional correctness.

For a given command P, we write $\mu_\bot$ for the memory that maps every variable of P to $\bot$. We let ρ range over distributions of memories, and write $\rho_\bot$ for the memory distribution that gives probability 1 to $\mu_\bot$. To any distribution ρ of memories whose domain includes X, we associate the distribution $\rho_{|X}$ of memories with domain X defined by $\rho_{|X}(\mu) = \sum_{\mu' \mid \mu'_{|X} = \mu} \rho(\mu')$ where $\mu'_{|X}$ is μ' restricted to X. Intuitively, $\rho_{|X}$ ignores the variables outside X, such as auxiliary variables introduced by the compilation. We write $\langle P, \rho\rangle$ for the configuration distribution d defined by $d(\langle P, \mu\rangle) = \rho(\mu)$ and $d(\langle Q, \mu\rangle) = 0$ for $Q \neq P$.

DEFINITION 5. *A compiler $\mathcal{C}$ is* correct *when, for any typable polynomial source program P, for $Q_0, \vec{Q} = \mathcal{C}(P)$, and for any polynomial core command P_0, we have*

$$\langle P_0; P, \rho_\bot\rangle \rightsquigarrow^* \langle \surd, \rho\rangle$$
$$\langle \mathit{next} := \mathfrak{start}; P_0; Q_0; N_n[\vec{Q}], \rho'_\bot\rangle \rightsquigarrow^* \langle \surd, \rho'\rangle$$

for memory distributions ρ and ρ' such that $\rho'_{|v(P_0;P)} = \rho$.

In the definition, P_0 initializes the source memory (much like $J; B_0$ or $J; B_1$) whereas Q_0 initializes auxiliary variables for the distributed code. Correctness states that, at least when the compiled code is scheduled using the correct scheduler N_n, our distributed implementation probabilistically simulates the source code on any input. (We believe that our correctness results would extend to any fair scheduler instead of N_n.)

To specify our security property, we need to define what is an active adversary against source commands. If P is a source program, let $\widehat{P}$ be the command context obtained from P by replacing every locality command of the form $b : P'$ with the command context $\_; P'; \_$. Intuitively, the additional holes in $\widehat{P}$ are placeholders for interleaving the code of an active adversary each time the implementation of P may yield as the result of distribution. (This is reminiscent of models of noninterference for concurrent programs, where the adversary may run between any two program steps.)

DEFINITION 6. *A compiler $\mathcal{C}$ is* computationally sound *when, for any adversary $\mathcal{A}$, for any safe source program P, for $Q_0, \vec{Q} = \mathcal{C}(P)$, for any polynomial commands J, B_0, B_1, for $X = C$ or I, if $\widehat{P}$ is CNI$_X$ for J, B_0, B_1 then $Q_0; \_[\vec{Q}]$ is CNI$_X$ for J, B_0, B_1.*

This security definition states that our implementation is at least as secure as the source program against active adversaries ($\widehat{P}$). Hence, if an adversary succeeds against our implementation, there must be an adversary that also succeeds against the source program.

4. CODE SLICING

The first stage of our compiler recursively slices a source command into a set of (core command) threads.

We distinguish between static threads (produced by slicing) and dynamic threads (their runtime instances). Threads in loops may be instantiated several times, so we need to distinguish between these instances to secure the control flow and prevent replay attacks. Hence, threads are parameterized by a tuple of *loop indexes*, treated as formal parameters for static threads and as distinct actual parameters for each of their runtime instances. We let $\mathtt{t}$ range over thread names and t range over dynamic thread identifiers, that is, pairs $\mathtt{t}\ \tilde{\imath}$ of a thread name and its tuple of loop indexes.

Threads outside of any loop are intended to run at most once in any execution of the program. Anticipating on the next section, this will be dynamically enforced by an anti-replay mechanism based on loop indexes, so for uniformity our identifiers always include a top-level index followed by an additional index for each nested loop. (In this paper, we formally consider a single execution of the program, so the top-level index is always 1, but more generally this index would be used to separate multiple executions of the program.) For instance, the identifier of a thread called from within two nested loops is of the form $\mathtt{t}\ i\,j\,k$, indicating the i^{th} execution of the program, the j^{th} execution of the outer loop and the k^{th} execution of the inner loop.

By convention, the initial thread is named start, and the final thread is named end. Thus, the static call graph is a finite directed graph between thread names, and the dynamic call graph is a (possibly infinite) directed acyclic graph between thread identifiers, with unique root $\mathsf{start}\ 1$ and a unique leaf $\mathsf{end}\ 1$.

From Source Programs to Local Threads Figure 2 specifies our slicing function $\mathcal{C}_1^-$ from source commands to thread definitions and core commands. Thread definitions are of the form

$$\textbf{thread}\ \mathtt{t}\ \tilde{\imath}\ (h, \iota) = \dot{S}$$

where $\mathtt{t}$ is a fresh thread name, $\tilde{\imath}$ is a tuple of loop indexes (with an index for every enclosing loop in source code), h is the local host to which the thread belongs, and ι is a fixed integrity level. For brevity, we sometimes write $\mathtt{t} : \dot{S}$ to refer to such a thread definition, and write $h(\mathtt{t})$ to access its host h.

The grammar for thread bodies after slicing ($\dot{S}$) is given below; S ranges over core commands (without localities).

$$\begin{array}{lcl} E & ::= & \textbf{call}\ \mathsf{t} \mid \textbf{goto}\ \mathsf{t} \mid \textbf{if}\ e\ \textbf{then}\ E\ \textbf{else}\ E \\ \dot{S} & ::= & S\textbf{;}\,\dot{S} \mid E \end{array}$$

Syntactically, a thread is a program for which execution ends with an auxiliary command **call** t or **goto** t. Command **call** t indicates that the next thread to execute belongs to another host, whereas **goto** t indicates that the next thread is local. We say that a thread $\mathtt{t}_0$ remotely (resp. locally) calls $\mathtt{t}_1$ when the body of $\mathtt{t}_0$ includes a command of the form **call** $\mathtt{t}_1\ \tilde{\imath}$ (resp. **goto** $\mathtt{t}_1\ \tilde{\imath}$), and that $\mathtt{t}_0$ is reachable from $\mathtt{t}_1$ when there is a possibly-empty series of calls (a *path*) from one to the other. A path is *local* when it contains only local calls.

Slicing helps ensure that every thread that is remotely called can locally compute the expected identifier of its caller. To this end, whenever a thread can call more than one thread (a branch) or can be called by more than one thread (a join), slicing ensures that the

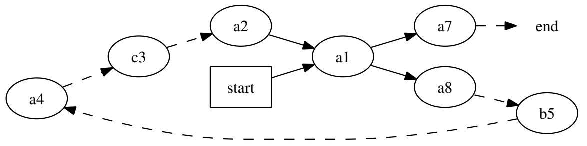
$$\begin{array}{l}
\mathcal{C}_1^-(h\colon S^+) \doteq \\
\quad \textbf{thread}\ \mathsf{start}\ i\ (h, I(S^+)) = \mathcal{C}_1'(i\ (h, I(S^+)), S^+)\ [\textbf{call}\ \mathsf{end}\ i] \\[6pt]
\mathcal{C}_1'(\tilde{\imath}\ (h,\iota), S)\ [\_] \doteq S;\_ \qquad\qquad \text{when } \mathsf{local}(S)\text{; otherwise:} \\
\mathcal{C}_1'(\tilde{\imath}\ (h,\iota), h'\colon S^+)\ [\_] \doteq \\
\quad \textbf{thread}\ \mathtt{t}\ \tilde{\imath}\ (h,\iota) = \_ \\
\quad \textbf{thread}\ \mathtt{t}'\ \tilde{\imath}\ (h', I(S^+)) = \mathcal{C}_1'(\tilde{\imath}\ (h', I(S^+)), S^+)\ [\textbf{call}\ \mathtt{t}\ \tilde{\imath}] \\
\quad \textbf{call}\ \mathtt{t}'\ \tilde{\imath} \\[6pt]
\mathcal{C}_1'(\tilde{\imath}\ (h,\iota), \textbf{if}\ e\ \textbf{then}\ {S^+}_0\ \textbf{else}\ {S^+}_1)\ [\_] \doteq \\
\quad \textbf{thread}\ \mathtt{t}\ \tilde{\imath}\ (h,\iota) = \_ \\
\quad \textbf{thread}\ \mathtt{t}_0\ \tilde{\imath}\ (h,\iota) = \mathcal{C}_1'(\tilde{\imath}\ (h,\iota), {S^+}_0)\ [\textbf{goto}\ \mathtt{t}\ \tilde{\imath}] \\
\quad \textbf{thread}\ \mathtt{t}_1\ \tilde{\imath}\ (h,\iota) = \mathcal{C}_1'(\tilde{\imath}\ (h,\iota), {S^+}_1)\ [\textbf{goto}\ \mathtt{t}\ \tilde{\imath}] \\
\quad \textbf{if}\ e\ \textbf{then goto}\ \mathtt{t}_0\ \tilde{\imath}\ \textbf{else goto}\ \mathtt{t}_1\ \tilde{\imath} \\[6pt]
\mathcal{C}_1'(\tilde{\imath}\ (h,\iota), \textbf{while}\ e\ \textbf{do}\ S^+)\ [\_] \doteq \\
\quad \textbf{thread}\ \mathtt{t}\ \tilde{\imath}\ (h,\iota) = \_ \\
\quad \textbf{thread}\ \mathtt{t}_t\ \tilde{\imath}\ j\ (h,\iota) = \textbf{if}\ e\ \textbf{then goto}\ \mathtt{t}_b\ \tilde{\imath}\ j\ \textbf{else goto}\ \mathtt{t}\ \tilde{\imath} \\
\quad \textbf{thread}\ \mathtt{t}_b\ \tilde{\imath}\ j\ (h,\iota) = \mathcal{C}_1'(\tilde{\imath}\ j\ (h,\iota), S^+)\ [\textbf{goto}\ \mathtt{t}_t\ \tilde{\imath}\ (j+1)] \\
\quad \textbf{goto}\ \mathtt{t}_t\ \tilde{\imath}\ 1 \\[6pt]
\mathcal{C}_1'(\tilde{\imath}\ (h,\iota), S_1^+\textbf{;}\,S_2^+)\ [\_] \doteq \mathcal{C}_1'(\tilde{\imath}\ (h,\iota), S_1^+)\ [\mathcal{C}_1'(\tilde{\imath}\ (h,\iota), S_2^+)\ [\_]]
\end{array}$$

Figure 2: Slicing algorithm

call is a goto, the callers and callees are all on the same host and, moreover, the matching branches and joins are on the same host.

Every thread is associated with an integrity level (used in the control flow protocol of Section 5). This level is computed by an overloaded function I, depending on the enclosing source locality $a : S^+$; it is defined as the greatest lower bound of the integrity levels of the written variables and declassifications of S^+.

EXAMPLE 2. *Example 1 yields 8 threads after slicing: 6 for a and 1 for b and c, with the following call graph*



The code of thread a1 i j is

if $y_{\text{LH}} < 3$ **then** {**goto**(*a8 i j*)} **else** {**goto**(*a7 i*)}

and the code of thread b5 i j is

if (y_{LH} *mod* 2) = 1
then {x_{HL} := $x_{\text{HL}} + 9$}
else {**skip**};
call(*a4 i j*)

Grouping Threads into Host Commands The commands **call** t and **goto** t are introduced only to keep track of the control flow during compilation; they are implemented as assignments to the variable next that holds the identifier of the next thread to execute:

$$\textbf{call}\ \mathsf{t} \doteq \textbf{goto}\ \mathsf{t} \doteq \mathit{next} \mathrel{\textbf{:=}} \mathsf{t}$$

Accordingly, to every thread definition $\textbf{thread}\ \mathtt{t}\ \tilde{\imath}\ (h,\iota) = \dot{S}$, we associate the command context

$$\mathit{Case}\,\mathtt{t}[\_] \doteq \textbf{if}\ \mathit{fst}(\mathit{next}) = \mathtt{t}\ \textbf{then}\ \dot{S}\{\mathit{snd}(\mathit{next})/\tilde{\imath}\}\ \textbf{else}\ \_$$

and, for every series of threads named $\mathtt{t}_0, \tilde{\mathtt{t}}$, we define

$$\mathit{Case}\,(\mathtt{t}_0, \tilde{\mathtt{t}})[\_] \doteq \mathit{Case}\,\mathtt{t}_0[\mathit{Case}\,\tilde{\mathtt{t}}[\_]]$$

After slicing, we regroup each remotely-callable thread together with all its locally-called threads and a local scheduler, as follows.

DEFINITION 7 (SCHEDULING). *The scheduling transformation $\mathcal{C}^+$ maps a series of thread definitions to a series of core commands $Q_{\mathtt{t}}$, one for each definition* **thread** $\mathtt{t}\ \tilde{\imath}\ (h,\iota) = \dot{S}$ *such that* $\mathtt{t}$ *is remotely called (starting with* $\mathtt{t} = \mathsf{start}$*), defined by*

$$\begin{aligned} Q_{\mathtt{t}} \ \doteq\ & \textbf{if } \mathit{fst}(\mathit{next}) = \mathtt{t} \textbf{ then } \{ \\ & \dot{S}\{\mathit{snd}(\mathit{next})/\tilde{\imath}\}\textbf{;} \\ & \textbf{while } \mathit{fst}(\mathit{next}) \in \tilde{\mathtt{t}} \textbf{ do } \mathit{Case}\,\tilde{\mathtt{t}}[\textbf{skip}]\} \end{aligned}$$

where $\tilde{\mathtt{t}}$ *collect the names of all threads locally reachable from* $\mathtt{t}$.

Finally, the compiler $\mathcal{C}_1$ produces $\vec{Q}$ by applying $\mathcal{C}_1^-$ followed by $\mathcal{C}^+$ and sets an empty initialization command ($Q_0 = \textbf{skip}$); the resulting code can be scheduled using $\mathit{next} \mathbin{\textbf{:=}} \mathsf{start}\ 1; N_n[\vec{Q}]$ with $\mathtt{stop} \doteq \mathsf{end}\ 1$. As can be expected at this stage of the compilation, we have correctness but not security, since the adversary may schedule our thread commands at will.

THEOREM 1 (SLICING). *$\mathcal{C}_1$ is correct.*

5. ENFORCING THE CONTROL FLOW

Next, we introduce program counters to keep track of the source control flow. For each integrity level ι associated with (at least) a thread, the variable pc_ι holds a thread identifier ($\mathtt{t}\ \tilde{\imath}$). At the beginning of each thread execution, for each integrity level ι, pc_ι identifies the last executed thread with integrity ι. We let $\Gamma(pc_\iota) = (\bot_C, \iota)$, $\Gamma(\mathit{last}_{\mathtt{t}}) = (\bot_C, I(\mathtt{t}))$, and $\Gamma(\mathit{next}) = (\bot_C, \top_I)$.

The body of each thread $\mathtt{t}\ \tilde{\imath}$ produced by the slicing algorithm is transformed by adding an assignment of the thread identifier to the corresponding program counter, by applying the command context

$$\mathrm{SPC}(\mathtt{t}\ \tilde{\imath})[\_] \ \doteq\ pc_{I(\mathtt{t})} \mathbin{\textbf{:=}} \mathtt{t}\ \tilde{\imath}\ \textbf{;}\ \_$$

We extend the definition of *Case* $\mathtt{t}[\_]$ accordingly.

To protect each remotely callable thread $\mathtt{t}$, we then guard its code using the command context $\mathrm{TPC}(\mathtt{t})$ below. (We use the command **check** e **then** S as syntactic sugar for **if** e **then** S **else skip**, to emphasize that a test failure should be interpreted as a global, silent runtime failure; see also Fournet and Rezk 2008.)

$$\begin{aligned} \mathrm{TPC}(\mathsf{start}\ i)[\_] \ &\doteq\ \textbf{check } \mathit{last}_{\mathsf{start}} = 0 \textbf{ then } \{\mathit{last}_{\mathsf{start}} \mathbin{\textbf{:=}} 1\textbf{;}\ \_\} \\ \mathrm{TPC}(\mathtt{t}\ \tilde{\imath})[\_] \ &\doteq\ \textbf{check } \textstyle\bigwedge_{\mathtt{t}'\ \tilde{\imath}' \in \mathrm{VA}(\mathtt{t}\ \tilde{\imath})} (pc_{I(\mathtt{t}')} = \mathtt{t}'\ \tilde{\imath}') \textbf{ then} \\ & \qquad \textbf{check } \mathit{last}_{\mathtt{t}} < \tilde{\imath} \textbf{ then } \{\mathit{last}_{\mathtt{t}} \mathbin{\textbf{:=}} \tilde{\imath}\textbf{;}\ \_\} \end{aligned}$$

where $I(\mathtt{t})$ is the integrity level associated to thread $\mathtt{t}$ and $\mathrm{VA}(\mathtt{t})$ is the set of thread identifiers t_0 such that there exists a (non-empty) path from t_0 to t for which all intermediate threads t' are such that:

$$I(\mathsf{t}') \not\leq_I I(\mathsf{t}_0) \quad \wedge \quad I(\mathsf{t}') \not\leq_I \textstyle\bigsqcap_{\mathfrak{s} \text{ locally reachable from } \mathsf{t}} I(\mathfrak{s})$$

In a trusted environment, the predicate checked by $\mathrm{TPC}(\mathsf{t})$ holds only when t is the next thread to execute. This predicate verifies the program counters for all predecessor threads, unless they have already been verified by a trusted predecessor. The test and the conditional assignment on variable $\mathit{last}_{\mathsf{t}}$ guarantee that the thread runs at most once for each tuple of loop indexes.

DEFINITION 8 (CONTROL FLOW). *The transformation $\mathcal{C}_2^+$ maps a series of thread definitions to a series of core commands $Q_{\mathtt{t}}$, one for each definition* **thread** $\mathtt{t}\ \tilde{\imath}\ (h,\iota) = \dot{S}$ *such that* $\mathtt{t}$ *is remotely called (starting with* $\mathtt{t} = \mathsf{start}$*), defined by*

$$Q_t \ \doteq\ \mathrm{TPC}(\mathtt{t}\ \tilde{\imath})[\mathrm{SPC}(\mathtt{t}\ \tilde{\imath})[Q'_{\mathtt{t}}]]\{\mathit{snd}(\mathit{next})/\tilde{\imath}\}$$

where

$$Q'_{\mathtt{t}} \ \doteq\ \dot{S}\textbf{; while } \mathit{fst}(\mathit{next}) \in \tilde{\mathtt{t}} \textbf{ do } \mathit{Case}\,\tilde{\mathtt{t}}[\textbf{skip}]$$

and $\tilde{\mathtt{t}}$ *collect the names of all threads locally reachable from* $\mathtt{t}$.

We define $\mathcal{C}_2$, the two first stages of the compilation, as the transformation obtained by applying $\mathcal{C}_1^-;\mathcal{C}_2^+$ and initializing all last-index variables with lists of zeros:

$$Q_0 \ \doteq\ (\mathit{last}_{\mathtt{t}} \mathbin{\textbf{:=}} 0\textbf{;}\ )_{\mathtt{t} \text{ remotely callable}}$$

EXAMPLE 3. *After enforcing the control flow, the code for the thread b5 i j given in Example 2 becomes*

```
check $pc1_{LH}$ = ("a8", [$i$;$j$]) then {
  check $last\_b5_{LL}$ < [$i$;$j$] then {
    $last\_b5_{LL}$ := [$i$;$j$]; $pc2_{LL}$ := ("b5", [$i$;$j$]);
    if ($y_{LH}$ mod 2) = 1
    then {$x_{HL}$ := $x_{HL}$ + 9}
    else {skip};
    call($a4\ i\ j$) } }
```

THEOREM 2 (CONTROL FLOW INTEGRITY). *$\mathcal{C}_2$ is correct and computationally sound.*

6. FROM SHARED VARIABLES TO LOCAL REPLICAS

So far, the security of our compiled code relies on the sharing of protected variables between hosts, with an abstract policy that limits the access rights of the adversary. To eliminate this assumption, we must ensure that all variables shared between hosts are unprotected (formally, that they have security label $(\bot_C, \top_I)$, to enable any adversary to read and write them), and instead dynamically protect their content using cryptography.

To do so, we first implement each shared variable (x) using a series of local variables ($\mathsf{t}.x$, $\mathsf{t}'.x, \ldots$), which we name replicas, and we insert explicit transfers between these replicas ($\mathsf{t}'.x := \mathsf{t}.x$) depending on the data flow. This stage of the compilation is much like a single-static-assignment transform [Alpern et al., 1988, Rosen et al., 1988]. It syntactically guarantees that, whenever a variable is read, we know which thread last assigned a value to that variable. We leave the cryptographic protection of transfers between different hosts to Section 7, where we will use the name of each replica as a unique tag for authenticating remote transfers.

Local Replicas and Policies We write $\mathsf{t}.x$ for the replica of x "owned" by t. Our intent is that only t assigns $\mathsf{t}.x$, so we refer to t as the last writer for x. The only exception arises when a thread t locally calls a thread t' that has other callers. In that case, for each variable x whose last writer at t' would depend on the caller, each caller pushes its replica into $\mathsf{t}'.x$ before the local call.

We extend our security policy for replicas, as follows. Assume t is a thread at host a. We set

$$C(\mathsf{t}.x) \ \doteq\ C(x) \sqcap C(a) \qquad I(\mathsf{t}.x) \ \doteq\ I(x) \sqcup I(a)$$

In case t reads x but does not write it, $\mathsf{t}.x$ may still be locally written (for instance, after verifying a MAC), but with reduced integrity. In case t writes x but does not read it, $\mathsf{t}.x$ may still be locally read (for instance as a temporary variable prior to encryption), but with reduced confidentiality.

Static Single Remote Assignments The translation for expressions and commands within a given thread is given in Figure 3. The translation for expressions annotates each read variable with its last writer. We let ϕ range over assignment maps, that is, functions from source variables to one of their replicas. We write _ for intermediate, unused maps. We define $\phi' \subseteq \phi$ as $\mathrm{dom}(\phi') \subseteq \mathrm{dom}(\phi)$ and $\forall x \in \mathrm{dom}(\phi').\ \phi'(x) = \phi(x)$. The translation for commands takes as input an assignment map ϕ and returns an updated assignment map; $\phi_{\mathtt{t}}$ is the initial map used to translate thread $\mathtt{t}$.

$$\llbracket x \rrbracket\phi \doteq \phi(x)$$
$$\llbracket op(e_0,\dots,e_{n-1})\rrbracket\phi \doteq op(\llbracket e_0\rrbracket\phi,\dots,\llbracket e_{n-1}\rrbracket\phi)$$

$$\llbracket \textbf{skip} \rrbracket\phi \doteq \textbf{skip}, \phi$$

$$\llbracket x \mathbin{\textbf{:=}} e\rrbracket\phi \doteq \mathfrak{t}.x \mathbin{\textbf{:=}} \llbracket e\rrbracket\phi,\ \phi[x \mapsto \mathfrak{t}.x]$$

$$\begin{array}{l}
\llbracket \textbf{if}\ e\ \textbf{then}\ S_0\ \textbf{else}\ S_1\rrbracket\phi \doteq \\
\quad \text{let } S_0', \phi_0' = \llbracket S_0(; x \mathbin{\textbf{:=}} x)_{x\in \mathrm{wv}(S_1)\setminus \mathrm{wv}(S_0)}\rrbracket\phi \text{ in} \\
\quad \text{let } S_1', \phi_1' = \llbracket S_1(; x \mathbin{\textbf{:=}} x)_{x\in \mathrm{wv}(S_0)\setminus \mathrm{wv}(S_1)}\rrbracket\phi \text{ in} \\
\quad \textbf{if}\ \llbracket e\rrbracket\phi\ \textbf{then}\ S_0'\ \textbf{else}\ S_1', \phi_0'
\end{array}$$

$$\begin{array}{l}
\llbracket \textbf{while}\ e\ \textbf{do}\ S\rrbracket\phi \doteq \\
\quad \text{let } S^h, \phi^h = \llbracket (x \mathbin{\textbf{:=}} x;)_{x\in \mathrm{wv}(S)}\rrbracket\phi \text{ in} \\
\quad \text{let } S', \phi' = \llbracket S\rrbracket\phi^h \text{ in} \\
\quad S^h \textbf{;}\ \textbf{while}\ \llbracket e\rrbracket\phi^h\ \textbf{do}\ S', \phi'
\end{array}$$

$$\begin{array}{l}
\llbracket S_0 \textbf{;} \dot{S}_1\rrbracket\phi \doteq \\
\quad \text{let } S_0', \phi_0 = \llbracket S_0\rrbracket\phi \text{ in let } \dot{S}_1', \_ = \llbracket \dot{S}_1\rrbracket\phi_0 \text{ in } S_0' \textbf{;} \dot{S}_1', \_
\end{array}$$

$$\begin{array}{l}
\llbracket \textbf{if}\ e\ \textbf{then}\ \dot{S}_0\ \textbf{else}\ \dot{S}_1\rrbracket\phi \doteq \\
\quad \text{let } \dot{S}_0', \_ = \llbracket \dot{S}_0\rrbracket\phi \text{ in let } \dot{S}_1', \_ = \llbracket \dot{S}_1\rrbracket\phi \text{ in} \\
\quad \textbf{if}\ \llbracket e\rrbracket\phi\ \textbf{then}\ S_0'\ \textbf{else}\ \dot{S}_1', \_
\end{array}$$

$$\llbracket \textbf{check}\ e\ \textbf{then}\ \dot{S}\rrbracket\phi \doteq \text{let } \dot{S}', \_ = \llbracket \dot{S}\rrbracket\phi \text{ in } \textbf{check}\ \llbracket e\rrbracket\phi\ \textbf{then}\ \dot{S}', \_$$

$$\llbracket \textbf{call}\ \mathtt{t}'\ \tilde{\imath}'\rrbracket\phi \doteq \textbf{call}\ \mathtt{t}'\ \tilde{\imath}', \_ \qquad \text{and check } \phi_{\mathtt{t}'} \subseteq \phi$$

$$\begin{array}{l}
\llbracket \textbf{goto}\ \mathtt{t}'\ \tilde{\imath}'\rrbracket\phi \doteq \\
\quad \text{let } X = \{x \in \mathrm{dom}(\phi_{\mathtt{t}'}) \mid \phi(x) \neq \phi_{\mathtt{t}'}(x)\} \text{ in} \\
\quad (\phi_{\mathtt{t}'}(x) \mathbin{\textbf{:=}} \phi(x);)_{x\in X} \textbf{;}\ \textbf{goto}\ \mathtt{t}'\ \tilde{\imath}', \_ \\
\quad \text{and check } \phi_{\mathtt{t}'} \subseteq \phi[x \mapsto \phi_{\mathtt{t}'}(x)]_{x\in X}
\end{array}$$

Figure 3: Replication algorithm (within thread $\mathfrak{t}$)

The first four cases concern local subcommands. Assignments reset the last writer to the current thread. Branches insert extra assignments for the variables written only in one branch; this ensures that $\phi_0' = \phi_1'$. Similarly, loops insert extra assignments so that ϕ' after the loop does not depend on the number of iterations. The remaining cases concern thread commands. Calls record constraints on the translation of their target threads ($\phi_{t'}$ only needs to keep the live variables of ϕ). Local calls are more complex, as the callee t' may merge several caller threads; hence, each caller propagates writes to the callee's replicas.

In the translation, we treat loop indexes symbolically, that is, we translate $\mathtt{t}\ \tilde{\imath} : P$ once for all instances of $\tilde{\imath}$. Accordingly, we merge initial calls and iterations: if $\mathtt{t}$ is called both to enter the loop (with final loop index 1) and to iterate (with final loop index $j+1$), we merge the two functions $\phi_{\mathtt{t}\ \tilde{\imath}\,1}$ and $\phi_{\mathtt{t}\ \tilde{\imath}\,(j+1)}$ into a single function $\phi_{\mathtt{t}}$ for translating P.

The translation ensures that, for every reader, there is always a unique last writer; this entails the insertion of transfers between local variables when there is a merge; these transfers do not change the semantics, but they may not comply with the source policy.

Finally, the translation of the threads $\mathtt{t}\ \tilde{\imath} : \dot{P}$ obtained at the end of Section 5 consists of the threads $\mathtt{t}\ \tilde{\imath} : \llbracket \dot{P}\rrbracket\phi_{\mathtt{t}}$ for some global assignment map $\phi_{\mathtt{t}}$ that meets all the inclusion constraints checked by the replication algorithm.

Initial and Final Values for Variables For a given program, an *input variable* is a variable that may be read before being written. Initially, ϕ_0 maps every input variable x to itself. To save the need for an ad hoc protocol to distribute initial values to multiple hosts, our implementation assumes that each input variable is initially read by a single host. Otherwise, for instance for a source program $a : P; b : P'$ where both P and P' may read x before it is written, one needs to manually rewrite the code, for instance into $a : \{x := x; P\}; b : P'$. The implementation restriction above is met when, after replication, the initial variable x occurs at a single host. Formally, our theorems do not rely on this assumption.

The final replica for each variable x is given by ϕ_{end}, the assignment map computed when we reach the end of the top-level thread. For consistency with the definitions of Section 3, we formally apply ϕ_{end}^{-1} to the host commands obtained after the replication stage, so that the final value for x is stored in variable x after running the implementation.

EXAMPLE 4. *After replication, the command for thread b5 is*

```
check (a8 i j.pc1) = ("a8", [i; j]) then {
  check last_b5_LL < [i; j] then {
    last_b5_LL := [i; j];
    b5 i j.pc2 := ("b5", [i; j]);
    if ((a8 i j.y) mod 2) = 1
    then {b5 i j.x := (a1 i j.x) + 9}
    else {skip; b5 i j.x := a1 i j.x};
    call(a4 i j) } }
```

Let $\mathcal{C}_3$ extend the compilation function of Section 5 with replication before grouping the threads into host commands. We have

THEOREM 3 (SINGLE REMOTE ASSIGNMENTS). *$\mathcal{C}_3$ is correct and computationally sound.*

7. CRYPTOGRAPHIC PROTECTION

We add cryptographic operations as required by the policy Γ, first for confidentiality, then for integrity. We protect only entry threads (called from some other host) and exit calls (calling some other host). Let $\mathfrak{t}$ be the thread to protect and $\mathfrak{t}'$ its callee. Before **call** $\mathfrak{t}'$, we encrypt the variables $E_{\mathfrak{t},\mathfrak{t}'}$ then MAC the variables $S_{\mathfrak{t},\mathfrak{t}'}$. As we enter $\mathfrak{t}$, we verify the MACs for variables $V_{\mathfrak{t}}$ that may be read or MACed in this thread, or recursively in any thread it may locally call, then we decrypt variables $D_{\mathfrak{t}} \subseteq V_{\mathfrak{t}}$ that may be read.

Next, we explain how to compute these sets of variables, collecting constraints on the keys to use for these operations.

Encryption Transform For protecting $\mathfrak{t} : P$, we set

$$D_{\mathfrak{t}} \doteq \bigcup_{\mathfrak{u}:Q \text{ locally reachable from } \mathfrak{t}} \{\mathfrak{s}.x \in \mathrm{rv}(Q) \mid C(x) \neq \bot_C \wedge h(\mathfrak{s}) \neq h(\mathfrak{t})\}$$

when $\mathfrak{t}$ is callable, $\emptyset$ otherwise.

$$E_{\mathfrak{t},\mathfrak{t}'} \doteq \bigcup_{\mathfrak{u}:Q \text{ reachable from } \mathfrak{t}'} \{\mathfrak{s}.x \in D_{\mathfrak{u}} \mid \mathfrak{t} \text{ locally reachable from } \mathfrak{s}\}$$

when $\mathfrak{t}$ calls $\mathfrak{t}'$, $\emptyset$ otherwise.

Hence, a thread decrypts a variable whenever this variable is (1) locally read by its code or one of its local callees, (2) not public, and (3) potentially written by a remote host; and a thread encrypts a variable whenever it may be decrypted later with this thread as potential writer. The encryption transform rewrites every callable thread $\mathfrak{t}$ as follows:

- replace every **call** $\mathfrak{t}'$ in every locally reachable $\mathfrak{u} : Q$ with $E(E_{\mathfrak{u},\mathfrak{t}'})$; **call** $\mathfrak{t}'$;
- replace every $\mathfrak{u} : P$ with $\mathfrak{u} : P\{\mathfrak{t}.x/\mathfrak{s}.x\}_{\mathfrak{s}.x\in D_t}$;
- replace the resulting $\mathfrak{t} : P$ with $\mathfrak{t} : D(D_{\mathfrak{t}}); P$.

The command $D(D_{\mathfrak{t}})$ assigns local replicas $\mathfrak{t}.x$ for every $\mathfrak{s}.x$ in $D_{\mathfrak{t}}$ after reading and decrypting new shared variables, such as $\mathfrak{t}.x^e$ or $\mathfrak{t}.x\_y^e$, with (at least) the same integrity as $\mathfrak{t}.x$ and no confidentiality ($\bot_C$). Conversely, the command $E(E_{\mathfrak{t},\mathfrak{t}'})$ reads $E_{\mathfrak{t},\mathfrak{t}'}$, then encrypts and writes them into these new variables.

We will define D as a series of decryptions, after grouping $D_{\mathfrak{t}}$ into tuples of variables that (1) share the same encryption key, and (2) are always jointly decrypted.

After the encryption transform, all remote reads occur on public variables (except for the key variables of D and E). On the other hand, the transform does not affect any confidential variables assigned at the same host, or pc_ι variables, for instance.

Authentication Transform For protecting $\mathfrak{t} : P$, we set

$$V_\mathfrak{t} \doteq \bigcup_{\mathfrak{u}:Q \text{ locally reachable from } \mathfrak{t}} \{\mathfrak{s}.x \in \mathrm{rv}(Q) \mid I(x) \neq \top_I \wedge h(\mathfrak{s}) \neq h(\mathfrak{t})\}$$
$$\text{when } \mathfrak{t} \text{ is callable, } \emptyset \text{ otherwise}$$
$$S_{\mathfrak{t},\mathfrak{t}'} \doteq \bigcup_{\mathfrak{u}:Q \text{ reachable from } \mathfrak{t}'} \{\mathfrak{s}.x \in V_\mathfrak{u} \mid \mathfrak{t} \text{ locally reachable from } \mathfrak{s}\}$$
$$\text{when } \mathfrak{t} \text{ calls } \mathfrak{t}', \emptyset \text{ otherwise}$$

Hence, a thread dynamically verifies the integrity of any variable that is (1) locally read by its code or one of its local callees, (2) somewhat trusted, and (3) MACed by a remote host. ($V_\mathfrak{t}$ includes in particular the variables $\mathfrak{t}.x^e$ read in $D(D_\mathfrak{t})$, as well as the pc_ι variables; all these variables are public.) And a thread MACs a variable when it may be verified later with this thread as writer. The authentication transform rewrites every callable thread $\mathfrak{t}$ as follows:

- replace every **call** $\mathfrak{t}'$ in every locally reachable $\mathfrak{u} : Q$ with $S(S_{\mathfrak{u},\mathfrak{t}'})$; **call** $\mathfrak{t}'$;
- replace every $\mathfrak{u}: Q$ with $\mathfrak{u}: Q\{\mathfrak{t}.x/\mathfrak{s}.x\}_{\mathfrak{s}.x \in V_\mathfrak{t}}$;
- replace the resulting $\mathfrak{t} : P$ with $\mathfrak{t} : V(V_\mathfrak{t})[P]$.

The command S may need to generate a tuple of MACs in case there are mutually-distrusting verifiers.

We will define S as a serie of MAC computations and V as a serie of nested verifications, after grouping variables that (1) share the same MAC key, and (2) are always jointly MACed/verified. For instance, we can always use a single MAC for all variables signed by the caller and read only by the callee.

After the authentication transform, all remote reads occur on unprotected variables (except for the cryptographic key variables).

Cryptographic Commands Figure 4 provides an implementation of our cryptographic commands applied to a single variable. The implementation relies on standard system libraries for the cryptographic primitives. ($\mathcal{SE}$ and $\mathcal{SD}$ are for symmetric encryption and decryption, and $\mathcal{M}$ and $\mathcal{V}_\mathcal{M}$ are for MAC computation and verification, respectively.) It introduces auxiliary variables to hold cryptographic values:

- $\mathfrak{s}.x^e$ for the encrypted value of $\mathfrak{s}.x$, with confidentiality $\bot_C$ and integrity $I(\mathfrak{s}.x)$;
- $\mathfrak{s}.x^u_H$ and $\mathfrak{s}.x^m_H$ for the (public, tainted) values and MACs of $\mathfrak{s}.x$, with label $(\bot_C, \bot_I)$.

To deal with sets of variables, we iterate these commands after grouping variables into tuples. We also use similar, asymmetric variants of these commands for initial key distribution.

A MAC verification V is an unsafe transfer (since $\mathfrak{t}.x$ is trusted but $\mathfrak{s}.x^u$ is tainted) guarded by a dynamic verification of the MAC. The MACed value consists of the concatenation of the full thread identifier (with its loop indexes), a unique constant for the source variable, and the authenticated value. In the proof, the security assumption on MACs enables us to treat it instead as a safe transfer from $\mathfrak{s}.x$ to $\mathfrak{t}.x$, with overwhelming probability.

Similarly, for encryptions and decryptions, the security assumption enables us to replace encryptions of plaintexts with encryptions of 0 and to perform a remote lookup instead of a decryption. The resulting, "ideal" variant of the implementation is the formal basis for our security proofs.

$$\begin{aligned}
V(\mathfrak{s}.x)[\_] &\doteq \textbf{check } \mathcal{V}_\mathcal{M}(\mathfrak{s}\hat{}\text{`}.x.\text{'}\hat{}\mathfrak{s}.x^u, \mathfrak{s}.x^m_H, k^a_H) \textbf{ then } \{\mathfrak{t}.x := \mathfrak{s}.x^u; \_\} \\
&\quad \text{with key } k^a_H \text{ shared by } H \supseteq \{h(\mathfrak{t}), h(\mathfrak{s})\} \\
D(\mathfrak{s}.x) &\doteq \mathfrak{t}.x := \mathcal{SD}(\mathfrak{s}.x^e, k^e_H); \\
&\quad \text{with key shared by } H \supseteq \{h(\tilde{w}), h(\mathfrak{t})\} \\
E(\mathfrak{t}.x) &\doteq \mathfrak{t}.x^e := \mathcal{SE}(\mathfrak{t}.x, k^e_H); \\
&\quad \text{with key shared by } H \\
S(\mathfrak{t}.x) &\doteq \mathfrak{t}.x^u := \mathfrak{t}.x;\ \big(\mathfrak{t}.x^m_H := \mathcal{M}(\mathfrak{t}\hat{}\text{`}.x.\text{'}\hat{}\mathfrak{t}.x, k^a_H);\big)_{H \in \tilde{H}} \\
&\quad \text{with keys shared by } \tilde{H} \text{ such that} \\
&\quad \mathfrak{t}.x \in V_u \Rightarrow \exists H \in \tilde{H}. H \supseteq \{h(\mathfrak{t}), h(\mathfrak{u})\}
\end{aligned}$$

Figure 4: Symmetric cryptographic operations

Shared-Key Selection We now briefly explain how we manage the keys used in Figure 4. We assume some consistent selection of hosts H sharing the keys between V and S, and between D and E. The compiler introduces families of shared variables k^a_H and k^e_H for these keys. The level of a symmetric key k_H shared by the hosts $a \in H$ is $\ell_H \doteq (\bigsqcap_{a \in H} C(a), \bigsqcup_{a \in H} I(a))$. By definition, this is the most secure label that can be both read and written by any of these hosts. We need robustness for each label ℓ_H for which we allocate a key. Intuitively, an adversary that can write (resp. read) the key can also read (resp. write) anything protected at that level.

Figure 4 only expresses functionality and security constraints on the keys, leaving the choice of which key to allocate and use to the compiler. This choice has a significant impact on the runtime cost of cryptographic protection. For instance, when a host performs a series of encryptions, it is worth solving their constraints with a minimal number of keys, so that we can first group the variables to protect and perform fewer, larger encryptions. Besides, encryption and authentication with compatible sets of hosts should clearly be replaced with joint authenticated encryption. We leave these optimizations as future work. For the time being, our compiler uses simple heuristics to minimize the overall number of keys.

Let K^a and K^e be the sets of subsets of hosts for which the compiler has allocated an authentication key or an encryption key, respectively. Accordingly, we define a command that generates these keys before running our implementation code:

$$Q'_0 \doteq (k^a_H := \mathcal{G}_\mathcal{M}();)_{H \in K^a} (k^e_H := \mathcal{G}_{\mathcal{SE}}();)_{H \in K^e}$$

and add this command before the initial command Q_0 of the previous compilation stage.

Our main theorem relies on two standard computational cryptographic assumptions: we say that an encryption (resp. MAC) scheme (resp. MAC scheme) is *secure* when it is IND-CCA2 (resp. INT-CMA).

THEOREM 4 (CRYPTOGRAPHIC PROTECTION).
If the cryptographic schemes used in $\mathcal{C}_4$ are correct and secure and all labels for the security keys are robust, then $\mathcal{C}_4$ is correct and computationally sound.

8. EXPERIMENTAL RESULTS

The prototype compiler consists of 11,000 lines of ML code. It is parameterized by a module that defines the security lattice (we have coded simple lattices and variants of the DLM [Myers and Liskov, 1998]). It takes as input a program written in an extension of the source language of Section 2 (S^+). It applies the translations of Sections 4, 5, 6, and 7, and produces a source ML program that can then be compiled using the F# compiler, and executed using the .NET runtime environment. The produced code is in a subset of ML similar to the core commands of Section 2; the main syntactic difference is that we use ML references instead of shared variables, and thus emit `x := !y` instead of `x := y`.

The compiler handles different types of data such as booleans, integers, strings, lists and tuples. To increase expressiveness of the source language, the programmer may use any ML function such as `printf` as primitive. The replication algorithm of Section 6 assumes given an initial map $\phi_{\mathtt{t}}$ for each thread; it describes how to check their correctness but not how to construct them. Concretely, our compiler relies on a fixpoint computation on dominance frontiers to build $\phi_{\mathtt{t}}$ [Cytron et al., 1991]. The produced code uses standard cryptographic primitives from the .NET libraries: AES with fresh random IVs for symmetric encryption, HMACSHA1 for symmetric MACs, RSA-OAEP for asymmetric encryption, and FDH-RSA for signatures. The sharing of global, unprotected memory is implemented by sending its updated content when calling a thread on a remote host. Communications rely on TCP connections. For each host, distribution information (IP addresses, ports, and public keys) is retrieved at run-time from a trusted configuration file.

Figure 5 summarizes our experimental results. *LOC* gives the number of lines of code for source and compiled programs; *l/t* gives the numbers of locality commands and of threads after splitting (remotely callable threads plus local threads); *crypto* gives the number of encryption/decryption and MAC/verification statements emitted by the compiler; *keys* gives the number of symmetric encryption/-MAC keys they use; *runtime* gives total execution times in seconds (without/with cryptography).

Program `empty` is just a:{**skip**}. It gives the baseline execution time due to the testing environment (which feeds default values to programs using a pipe) and execution initialization (mainly reading a configuration file). Program `running` is our running example (Example 1). Program `rpc` corresponds to a program that loops 500 times to increment a shared protected variable at two different locations. The cryptographic overhead is due to 2000 symmetric encryptions and decryptions, and 4000 MAC computations and verifications (for the incremented variable and the *pc* variable of the the control flow protocol). Program `guess` implements a basic "guess a number" with three participants. Program `hospital` collects information from three different principals that are then declassified by a doctor for the patient. Program `taxes` considers a scenario where a TPM (trusted platform module, with the most trustworthy integrity) runs a tax calculation with secret information provided by the user and a tax company.

9. CONCLUSION

We show how to compile high-level programs with information-flow policies to distributed systems, with adequate cryptographic protection to preserve their confidentiality and integrity properties. We believe this approach provides a safer, more reliable alternative to custom cryptographic protocol design. Our prototype compiler is still a proof of concept, whose performance can be significantly improved. Nonetheless, experimental results suggest that the cryptographic overhead is on par with handwritten code.

Acknowledgments We thank Jérémy Planul, Andrei Sabelfeld, David Sands, Eugen Zălinescu, and anonymous reviewers for their comments.

Program	LOC		l/t	crypto		keys	runtime	
`empty`	2	102	1 (1+0)	0/0	0/0	0/0	1.59	1.63
`running`	18	464	3 (5+3)	2/2	4/4	1/2	1.58	1.71
`rpc`	11	321	2 (3+3)	2/2	4/4	1/1	1.63	2.58
`guess`	52	912	7 (13+3)	2/2	13/16	2/3	1.69	1.98
`hospital`	33	906	5 (9+0)	4/4	11/11	4/8	1.70	1.84
`taxes`	55	946	4 (7+2)	8/8	16/16	4/6	1.71	1.77

Figure 5: Experimental results

References

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

B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of values in programs. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 1–11, Jan. 1988.

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.

S. Chong and A. C. Myers. Decentralized robustness. In *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006)*, pages 242–256. IEEE Computer Society, 2006.

H. Comon-Lundh and V. Cortier. Computational soundness of observational equivalence. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 109–118. ACM, 2008.

R. Cytron, J. Ferrante, B. K. Rosen, and M. N. Wegman. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13 (4):451–490, Oct. 1991.

C. Fournet and T. Rezk. Cryptographically sound implementations for typed information-flow security. In *35th Symposium on Principles of Programming Languages (POPL'08)*, pages 323–335. ACM, Jan. 2008.

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

P. Laud. On the computational soundness of cryptographically-masked flows. In *35th Symposium on Principles of Programming Languages (POPL'08)*, pages 337–348. ACM Press, 2008.

A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *19th IEEE Symposium on Research in Security and Privacy (RSP)*, Oakland, California, May 1998.

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

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

B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *ACM Symposium on Principles of Programming Languages*, pages 12–27. ACM, Jan. 1988.

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

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

S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20(3):283–328, 2002.

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