Compiling Information-Flow Security to Minimal Trusted Computing Bases

Cédric Fournet[2,1] and Jérémy Planul[1]

[1] MSR-INRIA
[2] Microsoft Research

Abstract. Information-flow policies can express strong security requirements for programs run by distributed parties with different levels of trust. However, this security is hard to preserve as programs get compiled to distributed systems with (potentially) compromised machines. For instance, many programs involve computations too sensitive to be trusted to any of those machines. Also, many programs are not perfectly secure (non-interferent); as they selectively endorse and declassify information, their relative security becomes harder to preserve.
We develop a secure compiler for distributed information flows. To minimize trust assumptions, we rely on cryptographic protection, and we exploit hardware and software mechanisms available on modern architectures, such as secure boots, trusted platform modules, and remote attestation.
We present a security model for these mechanisms in an imperative language with dynamic code loading. We define program transformations to generate trusted virtual hosts and to run them on untrusted machines. We obtain confidentiality and integrity theorems under realistic assumptions, showing that the compiled distributed system is at least as secure as the source program.

1 Programming with TPMs

When designing or reviewing the security of a system, a first step is to identify its trusted computing base (TCB), that is, the set of components that need to be trusted to achieve a given level of security. For general-purpose networked machines, this set is large and complex; it includes the hardware, an operating system, a runtime environment and their libraries (maybe 10^8 LOCs overall) plus drivers, applications, and dynamically loaded code. This leads to a best-effort approach to security, at odds with formal verification, which provides strong guarantees only for smaller, simpler systems.

Minimal TCBs. Modern computer architectures provide hardware support for reducing TCBs and protecting privileged operations. Thus, most computers come bundled with some form of secure coprocessor with a dedicated secure instruction set—for example, most laptops now embed a Trusted Platform Module (TPM) (TCG, 2005) and many high-end processors feature a special *late launch* functionality (AMD's Secure Virtual Machine Architecture, 2005, and Intel's Trusted Execution Technology, 2009). These instructions can run a given piece of code in isolation, with strong code-based identity and privileged cryptographic operations, for instance to seal persistent state or to perform remote attestation. Such hardware mechanisms can greatly reduce the TCB of security applications, by removing the need to trust the host operating system and other applications, and thus help protect critical data and computations from malicious

To appear in the proceedings of Programming Languages and Systems,
20th European Symposium on Programming, ESOP 2011.

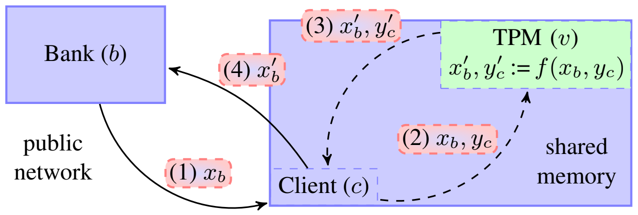
software. TPMs are routinely used for secure booting, e.g. BitLocker (Microsoft, 2006) guards access to the master keys for disk encryption, so that the disk content may be read only after authenticating the user and the operating system. Research papers also describe e.g. how to build secure online payment systems (Balfe and Paterson, 2008) and how to use late launches to run small pieces of application code in isolation (McCune et al., 2008, 2009). Still, the secure instructions are remarkably seldom used in practice. We believe that the complexity of their low-level interface and the lack of programming tools are major obstacles to their mainstream adoption for writing security applications.

Information-flow security. At a more abstract level, language-based security often relies on information flow policies (Denning, 1976; Myers and Liskov, 2000). Each variable is assigned a level in a security lattice; this level indicates the intended integrity and confidentiality of any information stored in this variable. Thus, a program is deemed completely secure (non-interferent) if an adversary that can access only low-level information cannot gain (or influence) any higher-level information by executing the program. Static analyses and type systems have been developed to verify that a program is secure with regards to a given policy. Further, it is sometimes possible to compile such programs to a given system while preserving their security properties. Hence, Jif (Myers et al., 2001) and FlowCaml (Pottier and Simonet, 2003) provide security typechecking for Java and Caml, respectively. Further, Jif/Split (Zdancewic et al., 2002; Zheng et al., 2003) and Swift (Chong et al., 2009) automatically partition distributed programs into local code, each running at a given security level, representing the level of trust granted to each host in a distributed system. As can be expected, program partitioning fails when no host is sufficiently trusted to run some parts of the computation, such as code that operates on secrets provided by mutually-suspicious parties. Cryptographically-blinded evaluation techniques (Diffie and Hellman, 1976) can sometimes solve this problem, but with a high performance overhead. Instead, in this paper, we systematically rely on secure hardware to virtually 'boot' short-lived, trusted environments for executing privileged code.

Example: applying for a loan. Consider a program involving two parties, a bank that offers loans, and a client that wishes to apply for a loan without disclosing private information (at least until the loan is granted). Suppose also that the bank does not want to disclose the parameters used for evaluating loan applications. Although the client and the bank do not trust one another, they may agree to securely run the loan-evaluation code on a TPM-enabled client machine. This simple computation is depicted below.

The bank sends its (encrypted, signed) secret input (x_b) to the client; the client forwards it to the code running the loan evaluation, together with its own input (y_c), using shared local memory; after securely booting, the TPM-protected code decrypts its input, evaluates the loan, and returns its results; finally, the client gets its result (y'_c) and may forward the (encrypted, signed) output (x'_b) to the bank if the loan is granted.

The messages passed between the bank and TPM-protected code must be cryptographically protected, so that for instance the bank input may be read and processed only by that code—not by the client or the network. The code protected by the TPM is short but complex as regards information flows: the inputs are endorsed (letting the client accept the bank input and vice versa) then the outputs are declassified (releas-



ing partial information from the client input to the bank and vice versa). Also, this code must refuse to run multiple loan evaluations for different client inputs without the bank's consent, as this may enable the extraction of the bank input.

Compiling with minimal TCBs. We compile imperative programs with security and locality annotations to distributed programs using cryptography and TPMs, and we show that our compilation scheme preserves information-flow security under standard cryptographic assumptions.

To this end, we specify a subset of the TPM instructions within a core imperative programming language. Our model aims at formal simplicity while still reflecting the main security features of hardware and cryptographic specifications, at a level of detail sufficient for reasoning about information flows. We model secure instructions to manage monotonic counters; measure code; run code in isolation; cryptographically sign data using the private attestation key of the TPM; and cryptographically seal and unseal data associated with some code.

We use this imperative language as the target of a new security-preserving compiler, built by adapting and extending recent work on cryptographic support for enforcing information-flow policies (Fournet and Rezk, 2008; Fournet et al., 2009). In their work, imperative commands are annotated with a host location, indicating where to run the command. Each location is also given a security level, used to type the source program. Their compiler, CFLOW, generates a protocol for securing the transfer of control between locations, as specified by the control flow of the source program, and selective encryption and authentication for securing the exchange of data.

We add support for dynamic code linking and a more permissive type system, enabling us to compile source programs that perform almost arbitrary declassifications and endorsements. We also provide runtime support for implementing highly-trusted locations by relying on secure instructions on relatively less trusted machines. Hence, we obtain distributed systems composed of ordinary application code and privileged code, with custom cryptographic support to coordinate their execution, such that all information-flow properties of the source program are preserved.

In summary, our main contributions are:

1. An operational semantics for modelling TPM-based security, focusing on TCB reduction by higher-order programming, with sample code and security properties.
2. A robust, flexible extension of CFLOW, enabling endorsement and declassification in typed source programs, with improved security definitions and theorems.
3. A compilation scheme for booting trusted hosts on demand, taking advantage of TPM attestation, with correctness and security theorems.

Contents. Section 2 defines an imperative, probabilistic, higher-order programming language. Section 3 defines information-flow policies, active adversaries, and target security properties. Section 4 describes and formalizes secure hardware instructions. Section 5 presents the CFLOW compiler and its theorems. Section 6 shows how to use TPM capabilities to implement secure virtual hosts, such as those produced by CFLOW.

An extended paper, the CFLOW compiler, and various code samples are available at **`http://msr-inria.inria.fr/projects/sec/cflow`**. The extended paper presents additional materials, including proofs; experimental results obtained by adapting CFLOW to generate statically-linked C code and running it on several small virtual machines; discussions on our shared memory model and scheduling; and additional results on attested boot sequences when the adversary can schedule, reboot, and corrupt host machines (but not their TPMs).

2 An imperative higher-order language

We define a probabilistic while-language with a command to turn data into executable code, used later to model dynamic code loading and TPM capabilities. The grammar for expressions and commands is

$$\begin{aligned} e &::= x \mid \mathit{op}(e_1, \ldots, e_n) \\ P &::= x := e \mid x := f(x_1, \ldots, x_n) \mid \mathit{skip} \mid P; P \\ &\quad \mid \text{if } e \text{ then } P \text{ else } P \mid \text{while } e \text{ do } P \mid \text{link}\, e\, [\widetilde{P}]\, \ell \mid X \end{aligned}$$

where *op* and f range over deterministic and probabilistic n-ary functions, respectively, with arity $n \geq 0$. Expressions e consist of variables and operations. We write *op* for nullary constructors *op*(). We assume given (polynomial-time) functions for standard boolean and arithmetic constants (0, 1, ...) and operators ($||$, $+$, ...). Commands P consist of variable assignments, using deterministic expressions and probabilistic functions, composed into sequences, conditionals, and loops, plus a link command for dynamically loading, linking, and running code. We write $\widetilde{P}$ for a tuple of commands $P_1, \ldots, P_n$ for some $n \geq 0$. Command variables X are placeholders for commands, bound in command contexts and when running link commands. As usual, we often use anonymous command variables in command contexts, writing $P[\widetilde{Q}]$ instead of $P[\widetilde{Q}/\widetilde{X}]$. For instance, $(X_1;X_2;X_1)[P_1,P_2]$ stands for $P_1;P_2;P_1$.

Commands as Data. We use data constructors to represent commands (and their expressions) as expressions, such as *op_if*(e_1,e_2,e_3) for conditionals and *op_x* for variable x. For instance, the command $c := c + 5$ is represented by *op_assign*(op_c, (*op_plus* (op_c,5))). To ease the writing of expressions representing commands, we let $\langle P \rangle$ be the expression that represents command P. Command expressions can also contain variables; these variables are quoted within $\langle P \rangle$. For instance, the expression *op_assign*(op_c, (*op_plus*(op_c,t))) is written $\langle c := c + \text{‘}t \rangle$.

The command link $e\, [\widetilde{P}]\, \ell$ dynamically checks that the result of expression e represents a valid command at level ℓ (the role of ℓ is explained below) parameterized by subcommand variables $\widetilde{P}$, and then runs that command after replacing each X_i with the command P_i. We write $\text{link}\, e\ \ell$ instead of $\text{link}\, e\, []\, \ell$ when $\widetilde{P}$ is empty. These checks occur at link-time, before running the command, as would be the case with a high-level virtual machine. In contrast, low-level protection for executable and data memory is usually enforced later, at runtime (e.g. by triggering memory page faults). Thus, for instance, information flows due to low-level memory error handling are outside the scope of our model. See also Askarov and Sabelfeld (2009) for a information-flow model of dynamic loading with run-time monitoring.

Probabilistic Semantics. The full paper details our operational semantics; in this presentation we only present our main notations. We use a probabilistic semantics 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. Our operational semantics is defined by a probabilistic reduction step relation $\rightsquigarrow_p$ between configurations, with $0 < p \leq 1$. We give below our rule for link commands (with $p = 1$).

$$\text{(LINK)}\quad \frac{[\![e]\!](\mu) = \langle P \rangle \qquad \vdash P : \ell}{\langle \text{link}\, e\, [\widetilde{P}]\, \ell, \mu \rangle \rightsquigarrow_1 \langle P[\widetilde{P}/\widetilde{X}], \mu \rangle}$$

We write $\Pr[\langle P, \mu \rangle; \varphi]$ for the probability that P terminates with a final memory that meets condition φ. When P always terminates, that is, $\Pr[\langle P, \mu \rangle; \textit{true}] = 1$, we write $\rho_\infty(\langle P, \mu \rangle)$ for the final distribution of memories obtained by running P with initial memory μ. For a given domain X, we write $\mu_{|X}$ for μ restricted to X and $\rho_{|X}$ for the projection of ρ on X, that is, $\rho_{|X}(\mu_{|X}) = \sum_{\mu' | \mu_{|X} = \mu'_{|X}} \rho(\mu')$.

Cryptographic assumptions. We consider only polynomial-time commands, and rely on standard computational definitions and assumptions for cryptography primitives; see the full paper for the details.

We use functions $\mathcal{G}_e$, $\mathcal{E}$, $\mathcal{D}$ and $\mathcal{G}_{\mathcal{SE}}$, $\mathcal{SE}$, $\mathcal{SD}$ for public-key and symmetric-key generation, encryption, and decryption; functions $\mathcal{G}_s()$, $\mathcal{S}$, and $\mathcal{V}$ for public-key generation, signing, and verification; functions $\mathcal{G}_\mathcal{M}$, $\mathcal{M}$, and $\mathcal{V}_\mathcal{M}$ for MAC key generation, computation, and verification; and functions $\mathcal{G}$ and $\mathcal{H}$ for pseudo-random hash function initialization and application.

3 Information flow security

Next, we define information-flow policies, we describe their enforcement by typing, and we discuss support for potentially-unsafe information flows. We then model active adversaries as command contexts, and give the general form of our target security properties.

Security labels. We annotate each variable with a security label. These labels specify the programmer's security intent. Except for dynamic links, they do not affect the operational semantics of programs. 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 $\sqcap$

$$\text{(TSubC)}\ \frac{\vdash P : \ell \qquad \ell' \leq \ell}{\vdash P :\ \ell'} \qquad \text{(TFun)}\ \frac{\vdash \widetilde{y} : \Gamma(x)}{\vdash x := f(\widetilde{y})\ : \Gamma(x)} \qquad \text{(TSeq)}\ \frac{\vdash P : \ell \qquad \vdash P' : \ell}{\vdash P; P'\ :\ \ell} \qquad \text{(TSkip)}\ \frac{}{\vdash \mathit{skip} : \top}$$

$$\text{(TCond)}\ \frac{\vdash e : \ell \qquad \vdash P : \ell \qquad \vdash P' : \ell}{\vdash \text{if } e \text{ then } P \text{ else } P'\ :\ \ell} \qquad \text{(TWhile)}\ \frac{\vdash e : \ell \qquad \vdash P\ :\ \ell}{\vdash \text{while } e \text{ do } P\ :\ \ell} \qquad \text{(TVar)}\ \frac{}{\vdash X : (\bot_C, \top_I)}$$

Strict rules:

$$\text{(TAssign Strict)}\ \frac{\vdash e : \Gamma(x)}{\vdash x := e : \Gamma(x)} \qquad \text{(TLink Strict)}\ \frac{\vdash e : \ell \qquad \vdash \widetilde{P} : (\bot_C, \top_I)}{\vdash \text{link}\, e\, [\widetilde{P}]\, \ell : \ell}$$

Lax rules:

$$\text{(TAssign Endorse)}\ \frac{\vdash e : (c, \_) \quad c \leq C(x)}{\vdash x := e : \Gamma(x)} \qquad \text{(TAssign Robust)}\ \frac{\vdash e : (c, \_) \quad c \not\leq C(x)}{\vdash x := e : \Gamma(x) \sqcap (\top_C, R(c))} \qquad \text{(TLink Privileged)}\ \frac{\vdash e : \ell \quad \vdash \widetilde{P} : \ell \quad \ell \leq \ell'}{\vdash \text{link}\, e\, [\widetilde{P}]\, \ell' : \ell}$$

Fig. 1. Security type system (for a fixed policy Γ)

for the least upper bound and greatest lower bound of two elements of $\mathcal{L}$, respectively. We write $\bot_C$, $\bot_I$, $\top_C$, and $\top_I$ for the smallest and largest elements of $\mathcal{L}_C$ and $\mathcal{L}_I$, respectively. In examples, we often use a four-point lattice defined by *LH* < *HH* < *HL* and *LH* < *LL* < *HL*, where *LH* for instance is low-confidentiality high-integrity.

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 at least as confidential (can be read by fewer entities) and at most as 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 $I(\ell) \leq_I I(\ell')$. We overload $\leq_C$ and $\leq_I$, letting $\ell \leq_C \ell'$ be $C(\ell) \leq_C C(\ell')$ and $\ell \leq_I \ell'$ be $I(\ell) \leq_I I(\ell')$. We let $\ell^I \doteq (\bot_C, I(\ell))$ be the label with low confidentiality and the integrity of ℓ.

Policies. Memory policies are functions Γ from variables to security labels. We define *low equality* between memories, memory distributions, and distributions, relative to a label $\ell \in \mathcal{L}$: letting $S = \{x \mid \Gamma(x) \leq \ell\}$, we define $\mu =_\ell \mu'$ as $\mu_{|S} = \mu'_{|S}$, $\rho =_\ell \rho'$ as $\rho_{|S} = \rho'_{|S}$, and $d =_\ell d'$ as $d_{|S} = d'_{|S}$.

A strict type system for non-interference. As a starting point, we equip our language with a type system that enforces (termination-insensitive) non-interference. Typing judgments for commands are of the form $\Gamma \vdash P : \ell$. We often omit the policy Γ when it is clear from the context.

The typing rules for commands appear in Figure 1 (excluding the 'lax' rules). We omit the standard typing rules for expressions, such that $\vdash e : \ell$ when $\Gamma(x) \leq \ell$ for each variable x read in e. This type system is similar to those typically used for non-interference (see e.g. Sabelfeld and Myers, 2003). The only new rule is TLink Strict: the command $\text{link}\, e\, [\widetilde{P}]\, \ell$, when executed, will check that the expression e represents a valid command at level ℓ before running it. Accordingly, we type the command also at level ℓ, after checking that its actual auxiliary commands have level $(\bot_C, \top_I)$, as anticipated by rule TVAR. (We considered typing auxiliary commands and command variables at other levels, but this is not needed for our present purpose.) We also check that the expression e has level ℓ, to keep track of the implicit flow from command values to their runtime effects. To illustrate this rule, consider two variables $secret$ and x at levels $\ell_s = \Gamma(secret)$ and $\ell_x = \Gamma(x)$ such that $\ell_s \not\leq \ell_x$. We have:

1. *x* := *secret* is not typable.
2. $\vdash$ link $\langle$*x* := *secret*$\rangle$ ℓ : ℓ but runtime type checking will fail.
3. link $\langle$*x* := '*secret*$\rangle$ ℓ_x is not typable (preventing a flow from the command expression).
4. $\vdash$ link $\langle$*x* := '*secret*$\rangle$ ℓ_s : ℓ_s but runtime type checking will fail.
5. link $\langle$if *secret* then *X*$\rangle$[*x* := 0] ℓ_s is not typable unless $\ell_x = (\bot_C, \top_I)$.
6. $\vdash$ link $\langle$if *secret* then *X*$\rangle$[*x* := 0] ℓ_s : ℓ_s if $\ell_x = (\bot_C, \top_I)$ but runtime type checking will also fail (preventing an implicit flow from running the auxiliary command).

With the strict typing rules, typing guarantees non-interference:

Theorem 1 (Non-interference). *Let Γ be a security policy, P a (strictly) well-typed command, $\ell \in \mathcal{L}$, and μ_0 and μ_1 two initial memories such that P always terminates.*
If $\mu_0 =_\ell \mu_1$, then $\rho_\infty(\langle P, \mu_0\rangle) =_\ell \rho_\infty(\langle P, \mu_1\rangle)$.

Declassifications and endorsements. Our strict type system thus excludes many useful programs that (by design) selectively declassify secrets or endorse untrusted values.

Example 1 (Password protection). Consider a program that releases a secret after verifying a password entered in variable *guess*:

if *guess* = *pwd* then *r* := *secret*

with $\Gamma(guess)$ = *LL*, $\Gamma(pwd)$ = *HH*, $\Gamma(r)$ = *LH*, and $\Gamma(secret)$ = *HH*. Although this program is arguably secure, it endorses *guess* (which is a priori untrusted), declassifies the outcome of the test (which is secret, since *pwd* is), then possibly declassifies $secret$.

Consider now a system that provides a subcommand that conditionally releases a secret after verifying a password, and tolerates up to three failed attempts (to protect against brute-force attacks on the password):

c := 0;
link *a*[if $c < 3$ && *guess* = *pwd* then *r* := *secret* else *c*++] *LL*

using a counter variable c with $\Gamma(c)$ = *LH* and a variable a with $\Gamma(a)$ = *LL* for dynamically loading code that may call this subcommand. Intuitively, a contains arbitrary low-level code, representing an active adversary, that cannot leak *pwd* or *secret* and cannot write *pwd*, *secret*, or *c*. If linking succeeds, this code can only access the secret by calling its privileged subcommand, and only the first three calls may succeed. For instance, running the command above with an initial memory where a is set to the command *guess* := 0; while *guess* < 10 && *r* = 0 do { *X*; *guess*++ }

leaks *secret* to r only if *pwd* is 0, 1, or 2. More generally, if we also assume that *pwd* is sampled at random, the probability that any adversary command learns anything about *secret* is bounded by the sum of the probabilities of the three most probable passwords ($3/N$ if there are N uniformly distributed passwords).

We intend to compile such programs, letting the programmer take responsibility for the source properties of her program, but still ensuring that the compilation process does not introduce any further potentially-unsafe information flows.

Robustness. In the example above, the programmer deliberately declassifies information. Moreover, the declassification depends on the low-integrity variable *guess*, thereby letting the adversary influence what is declassified. This is generally dangerous; for instance, our example would be entirely broken with $\Gamma(pwd) = HL$. Conversely, a declassification is *robust* when it does not depend on low-integrity data, and thus cannot be influenced by active adversaries (Zdancewic and Myers, 2001; Sabelfeld and Myers, 2004; Chong and Myers, 2006; Askarov and Myers, 2010). We support non-robust declassifications, treating them as a high-integrity endorsement followed by a robust declassification, and we rely on a *robustness function*, $R : \mathcal{L}_C \to \mathcal{L}_I$, that indicates the minimum integrity level required to declassify each confidentiality level. Although we allow endorsement and non-robust declassification, in the following, we still usually demand that our security policies be robust:

Definition 1 (Robust policies). *For a given robustness function R, $\ell \in \mathcal{L}$ is robust when $I(\ell) \leq R(C(\ell))$; Γ is robust for R when $\Gamma(x)$ is robust for all $x \in \mathrm{dom}(\Gamma)$.*

A more permissive type system. For typing source programs, we define a new type system, whose typing rules allow declassifications and endorsements but take them into account to compute the level of the command (sometimes called its 'program counter' level). The typing rules appear in Figure 1, using three new 'lax' typing rules instead of the 'strict' ones. The rules TASSIGN ENDORSE and TASSIGN ROBUST are two generalization of rule TASSIGN STRICT. TASSIGN ENDORSE is TASSIGN STRICT only when e has at least the integrity of x; otherwise, the assignment endorses e. Irrespective of e, the command is typed at the level of x. TASSIGN ROBUST enables the declassification of e into x ($c \not\leq C(x)$) but it records this privileged operation by raising the command type up to the associated robust integrity level $R(c)$. The rule TLINK PRIVILEGED generalizes TLINK STRICT by allowing the caller to link e with auxiliary commands at arbitrary levels of integrity, but it records those levels in the type of the command. This endorses at link-time any calls to the auxiliary commands, since dynamic typing of the callee ignores the integrity of auxiliary command variables (rule TVAR).

The lax type system enforces two fundamental properties. First, if a command has level ℓ, then it does not write variables below ℓ. Second, a command at level ℓ may declassify values of confidentiality c only if $I(\ell) \leq R(c)$.

Theorem 2 (Containment). *Let Γ be a policy, $\ell \in \mathcal{L}$, P a command, and μ a memory such that P terminates. If $\vdash P : \ell'$ and $\ell' \not\leq \ell$, then $\rho_\infty(\langle P, \mu\rangle) =_\ell \mu$.*

Theorem 3 (Robust non-interference). *Let Γ be a policy, $\ell \in \mathcal{L}$, $c \in \mathcal{L}_C$ such that $I(\ell) \not\leq R(c)$, and $L = \{x \mid c \not\leq C(x)\}$. Let P be a command and μ_0, μ_1 memories such that P terminates. If $\vdash P : \ell$ and $\mu_{0|L} = \mu_{1|L}$, then $\rho_\infty(\langle P, \mu_0\rangle)_{|L} = \rho_\infty(\langle P, \mu_1\rangle)_{|L}$.*

Active adversaries. In the following, our security properties are parameterized by the power of the adversary, defined by a security level $\alpha \in \mathcal{L}$. We say that a command A is an adversary command (respectively an adversary command context) if it reads only variables of lower confidentiality than $C(\alpha)$, write only variables of higher integrity than $I(\alpha)$, and use links only with a level above α. We are interested in robustness functions that ensure adversaries can read any variable they can write, at least when the policy Γ is robust.

Definition 2. *R is robust against an adversary level α when, for all confidentiality levels $c \in \mathcal{L}_C$, if $I(\alpha) \leq R(c)$, then $c \leq C(\alpha)$.*

Our next theorem guarantees that, with R robust against α and with a robust security policy, an adversary command does not gain additional expressiveness by using link. This justifies our condition on link in the definition of adversary commands.

Theorem 4. *Let α be an adversary level, R be a robustness function robust against α, Γ be a security policy robust for R, and A an adversary command. There exists an adversary command A' with no link commands such that, for all memory μ, if A terminates on μ, then A' terminates on μ and $\rho_\infty(\langle A, \mu\rangle) = \rho_\infty(\langle A', \mu\rangle)$.*

Properties of program transformations. We finally present the main properties we establish for our compiler, as regards both security and functional correctness. The properties are stated below for an abstract program transformation between source and target programs; they are instantiated before each of our main theorems in Sections 5 and 6.

We are interested in transformations that operate on programs that are (possibly) not perfectly secure, so we cannot define security as the preservation of non-interference. Instead, for each of our transformations, we demand that there is an inverse map from target adversaries to source adversaries, essentially showing how to 'decompile' each attack into an attack already present before the transformation is applied.

We consider system configurations obtained by composing an imperative program $P \in \mathcal{P}$, an adversary (e.g. a command context) $A \in \mathcal{A}$, and an initial state (e.g. a memory) $\mu \in \mathcal{M}$. Their semantics is given by an evaluation function, written $\langle\!\langle \cdot \rangle\!\rangle : \mathcal{P} \to \mathcal{A} \to \mathcal{M} \to \mathcal{M}$, and an observational equivalence on states, written $\approx \subseteq \mathcal{M}^2$. For a given program and an arbitrary adversary, we are interested in the properties of final states up to $\approx$. For instance, if we consider commands for a client and a bank scheduled by an adversary that controls the network, we may let programs range over pairs of commands Q_c, Q_b, let adversaries range over binary command contexts, let $\approx$ be low-equality on memory distributions, and use the evaluation function

$$\langle\!\langle (Q_c, Q_b), A, \mu \rangle\!\rangle \ \doteq\ \rho_\infty(\langle A[Q_c, Q_b], \mu\rangle)$$

Given definitions for source $(\mathcal{P}, \mathcal{A}, \mathcal{M}, \approx, \langle\!\langle\rangle\!\rangle)$ and target $(\mathcal{P}', \mathcal{A}', \mathcal{M}', \approx', \langle\!\langle\rangle\!\rangle')$ configurations, we consider *program transformations*, written $[\![\cdot]\!] : \mathcal{P} \to \mathcal{P}'$, together with *state projections* (that is, surjective functions), written $\pi : \mathcal{M}' \to \mathcal{M}$. (The role of π is to erase any auxiliary variable introduced by the transformation.) We arrive at the following definitions:

Definition 3 (Security). $([\![\cdot]\!], \pi)$ *is secure when, for every source program* $P \in \mathcal{P}$ *and target adversary* $A' \in \mathcal{A}'$*, there is a source adversary* $A \in \mathcal{A}$ *such that, for every target initial memory* $\mu' \in \mathcal{M}'$*, we have* $\langle\!\langle P, A, \pi(\mu') \rangle\!\rangle \approx \pi \langle\!\langle [\![P]\!], A', \mu' \rangle\!\rangle'$.

Definition 4 (Correctness). $([\![\cdot]\!], \pi)$ *is correct when, for every source program* $P \in \mathcal{P}$ *and source adversary* $A \in \mathcal{A}$*, there is a target adversary* $A' \in \mathcal{A}'$ *such that, for every target initial memory* $\mu' \in \mathcal{M}'$*, we have* $\langle\!\langle P, A, \pi(\mu') \rangle\!\rangle \approx \pi \langle\!\langle [\![P]\!], A', \mu' \rangle\!\rangle'$.

The two definitions differ in their quantification on adversaries. Informally, all attacks must be reflected, but not all of them need to be preserved, so we expect functional correctness only for a well-behaved subset of adversaries, acting for instance as reliable networks and fair schedulers, and we will use a smaller set $\mathcal{A}$ for Definition 4 than for Definition 3.

4 Command semantics for secure instructions

We model a core subset of the security features available on modern processors with a TPM. Aiming at formal simplicity, we do not account for all the details of their hardware specification, but still intend to reflect their gist. We set an (intuitively high) robust security level ℓ_{TPM} for the hardware, and assign that level to fixed variables that model parts of the hardware-protected memory, together with fixed commands that model secure instructions and have privileged access to these variables. (Their initialization is described at the end of the section.) We then model software as commands linked to these privileged subcommands, thereby gaining indirect access to protected variables.

Related work. We briefly discuss prior models and analyses for TPMs. Abadi and Wobber (2004) give an authorization logic for a precursor of the TPM. Gürgens et al. (2008) analyze several TCG protocols. Millen et al. (2007) study remote attestation using a model-checker. Datta et al. (2009) develop a logic for reasoning about attestation and secure boots. Our model of the TPM differs from theirs in its use of information flows, memory policies, and cryptographic assumptions. It also covers confidentiality properties and deals with sealing and unsealing.

Monotonic counters. The TPM features a collection of monotonic counters, that is, persistent protected memory whose contents can only be read and incremented, but not reset (TCG, 2006, p 681). Such counters are essential for protection against replays.

We model just one of these counters, using a public variable c at the integrity level of the TPM. Thus, our counter can be read by any command but it is exclusively assigned by the fixed command INC below. In particular, c cannot be reset or decremented.

$$\mathrm{INC} \doteq \quad c := c{+}1 \qquad\qquad \Gamma(c) = \ell^{I}_{TPM}$$

(Concretely, TPMs manage a few independent counters with finer access control, and the operating system is in charge of restricting increments to prevent denial of service.)

Example 2. Continuing with our password example, we may use the monotonic counter to reliably keep track of guessing attempts:

$$\text{link } a[\text{ if } c < 3 \text{ \&\& } \mathit{guess} = \mathit{pwd} \text{ then } r := \mathit{secret} \text{ else INC }]\ \mathit{LL}$$

Platform configuration registers. The TPM also features a collection of Platform Configuration Registers (PCR), which are cleared when the machine reboots, then selectively written by TPM commands. As detailed in the full paper, these registers usually contain measurements of the code running on the machine. PCRs are specialized: the first PCRs are used for static root of trust measurements as the machine boots (SRTM) (Grawrock, 2007), while PCRs 17–19 are used for dynamic root of trust measurements (DRTM) and may be selectively reset without rebooting (TCG, 2006; AMD, 2005). PCRs are read as high-integrity implicit parameters for many other TPM commands, such as attestation and seals. We model PCRs as variables h_i at level ℓ^I_{TPM}. For simplicity, we use just two registers, h_1 for SRTM, modified only by EXTEND$_1$, and h_{17} for DRTM, initialized by SKINIT and modified by EXTEND$_{17}$, as explained below.

EXTEND$_i$ appends some code identity to h_i and can be used to record a delegation chain starting from a secure kernel (TCG, 2006, p 284). To keep the size of h_i constant, the chain is implemented as a nested hash, using a cryptographic hash functions $\mathcal{H}$. We model it as

$$\text{EXTEND}_i \doteq \quad h_i := \mathcal{H}(h_i|\mathit{identity}) \qquad \Gamma(h_i) = \ell^I_{\mathit{TPM}} \qquad \Gamma(\mathit{identity}) = (\bot_C, \top_I)$$

where '|' is bitstring concatenation and *identity* is a public-untrusted variable .

SKINIT sets the code identity in h_{17} to (the hash of) a new command passed as input, usually called a 'secure kernel', then runs that command, and finally clears h_{17} (AMD, 2005, p 53). We model it as an assignment to h_{17} from the content of a public-untrusted variable *kernel*, followed by a link of *kernel* with subcommands parameters that pass to the new kernel the rest of the TPM interface (written $\widetilde{\mathit{TPM}}$), then a reset of h_{17}. We let $\widetilde{\mathit{TPM}} \doteq \{$INC,EXTEND$_{17}$,ATTEST$_{17}$,SEAL$_{17}$,UNSEAL$_{17}\}$.

$$\text{SKINIT} \doteq \quad h_{17} := \mathcal{H}(kernel);\ \text{link } \mathit{kernel}[\widetilde{\mathit{TPM}}]\ \ell^I_{system};\ h_{17} := 0$$

Thus, h_{17} either is at its default value 0 or it holds the identity of the kernel that is currently running, possibly extended by a chain of hashes that records further identity information. Concretely, the command SKINIT loads code at a privileged (kernel) level. This is reflected in our model by the link label ℓ^I_{system}. It is the responsibility of the operating system to validate the kernels passed by user commands before calling SKINIT e.g. to prevent privilege escalation. In the following, we assume that the hash function used for all assignment to PCRs is collision-resistant and yields fixed-sized hash values (so that concatenation of a hash with another value is injective).

Remote attestation. Each TPM uses a fixed public-key-signature keypair, set during manufacturing, and used to uniquely identify and authenticate this particular TPM.

ATTEST signs an input value and a subset of the PCRs with the private signing key (TCG, 2006). The resulting signature guarantees that this value has been 'attested' by a command running on a machine with this TPM and these PCR values. This signature can be verified by any command that knows the verification key for the TPM, typically running on a remote machine; the verifier can then interpret the authenticated value and PCRs. We model attestation with two variables for the TPM keypair, k^{+}_{TPM} of level ℓ^{I}_{TPM} and k^{-}_{TPM} of level ℓ_{TPM}, and two commands

$$\begin{aligned}
\text{ATTEST}_i &\doteq \quad \mathit{tag} := \mathcal{S}(i|h_i|\mathit{plain},\, k^{-}_{TPM})\\
\text{VERIFY}_i &\doteq \quad \text{if } \mathcal{V}(i|\mathit{source}|\mathit{plain},\, \mathit{tag},\, k^{+}_{TPM}) \text{ then } X
\end{aligned}$$

with public-untrusted inputs *source* and *plain* for the presumed value of h_i and the attested value, and output *tag* for the signature. These commands rely on public-key signing ($\mathcal{S}$) and signature-verification ($\mathcal{V}$) functions. Since the verification key is public, VERIFY need not be a privileged command; its variable X stands for the command guarded by the cryptographic verification.

Sealing. SEAL encrypts and signs the content of a variable together with the current identity of the sender and the intended identity of the receiver (TCG, 2006, p 298). Conversely UNSEAL decrypts a variable and verifies its signature, then verifies the identity of the sender and the current identity of the receiver (TCG, 2006, p 364). The TPM can handle several nonmigratable keys, but we only model sealing and unsealing keyed with fixed hardware secrets $s.ke$ and $s.ka$, both at level ℓ_{TPM}.

$$\begin{aligned}
\text{SEAL}_i \doteq \quad & \mathit{enc} := \mathcal{SE}(\mathit{plain},\mathit{s.ke});\ \mathit{mac} := \mathcal{M}(i|h_i|\mathit{target}|\mathit{enc},\mathit{s.ka});\\
& \mathit{cipher} := \mathit{enc}|\mathit{mac};\ \mathit{enc} := 0;\ \mathit{mac} := 0\\
\text{UNSEAL}_i \doteq \quad & \mathit{enc}|\mathit{mac} := \mathit{cipher};\\
& \text{if } \mathcal{V}_{\mathcal{M}}(i|\mathit{source}|h_i|\mathit{enc},\, \mathit{mac},\, \mathit{s.ka})\\
& \text{then } \mathit{plain} := \mathcal{SD}(\mathit{enc},\mathit{s.ke}) \text{ else } \mathit{plain} := 0;\\
& \mathit{enc} := 0;\ \mathit{mac} := 0
\end{aligned}$$

where $\mathit{enc}|\mathit{mac} := \mathit{cipher}$ is syntactic sugar for assigning to *enc* and *mac* substrings of *cipher* at fixed indexes (since the size of *mac* is fixed). As illustrated in the rest of the paper, SEAL and UNSEAL can be used to emulate a persistent, secure memory, and to communicate securely between TPM commands.

Security and functionality properties for seals. We specify the cryptographic properties of SEAL and UNSEAL by relating them to an ideal implementation that maintains a global table for all values sealed so far and encrypts 0s instead of the actual plaintexts. (The full paper define similar security and functionality properties for attestation.)

$$\begin{aligned}
\text{SEAL}^0_i \doteq \quad & \mathit{enc} := \mathcal{SE}(0,\mathit{s.ke});\ \mathit{mac} := \mathcal{M}(i|h_i|\mathit{target}|\mathit{enc},\mathit{s.ka});\\
& \mathit{cipher} := \mathit{enc}|\mathit{mac};\ \mathit{log}_i := \mathit{log}_i + ((h_i|\mathit{target}|\mathit{enc}),\mathit{plain})\\
& \mathit{enc} := 0;\ \mathit{mac} := 0\\
\text{UNSEAL}^0_i \doteq \quad & \text{if } \mathcal{V}_{\mathcal{M}}(i|\mathit{source}|h_i|\mathit{enc},\, \mathit{mac},\, \mathit{s.ka})\\
& \text{then } \mathit{plain} := \mathit{assoc}(\mathit{log}_i,(\mathit{source}|h_i|\mathit{enc})) \text{ else } \mathit{plain} := 0;\\
& \mathit{enc} := 0;\ \mathit{mac} := 0
\end{aligned}$$

Security means that, provided s is generated uniformly at random and no other part of the code accesses s or log, no probabilistic polynomial program can distinguish between (SEAL, UNSEAL) and (SEAL^0,UNSEAL^0). This property can be reduced to indistinguishability against chosen plaintext attacks for encryption and resistance against forgery attacks for signing.

Functionality means that UNSEAL is a partial inverse of SEAL: unsealing a value sealed with matching source and target hashes always yields the plain sealed value.

Auxiliary notations. For convenience, we define simple macros for calling SEAL and UNSEAL. (The full paper defines similar macros for our other security commands.)

$$x := \mathrm{SEAL}_i(e_v, e_t) \doteq \quad \mathit{target} := e_t;\ \mathit{plain} := e_v;\ \mathrm{SEAL}_i;\ x := \mathit{cipher};\ \mathit{target} := 0;\ \mathit{plain} := 0;\ \mathit{cipher} := 0$$

$$x := \mathrm{UNSEAL}_i(e_c, e_s) \doteq \quad \mathit{source} := e_s;\ \mathit{cipher} := e_c;\ \mathrm{UNSEAL}_i;\ x := \mathit{plain};\ \mathit{source} := 0;\ \mathit{cipher} := 0;\ \mathit{plain} := 0$$

Example 3. Continuing with our password example, we define code that seals the secret and the password to itself (using the current value of h_{17}) within a public-untrusted variable. Thus, protected by the TPM key, the secret and the password can only be retrieved by re-running this code. When re-run, the code behaves as in the password example, retrieving the password and the secret then granting access to the secret if the password is guessed in less than three attempts.

```
kernel := ⟨
  if c = 0 then store := SEAL((pwd,secret),h_17)
  else { pwd,secret := UNSEAL(store,h_17);
         if c < 4 && guess = pwd then r := secret};
  INC; pwd := 0; secret := 0 ⟩;
SKINIT;A [SKINIT]
```

Assuming that, initially, $c = 0$ and *pwd* is sampled at random, the probability that a polynomial adversary learns anything about *secret* is bounded by the sum of the probabilities of the three most probable passwords plus the (negligible) probabilities that the adversary finds a collision in the hash function or breaks the cryptography used in SEAL and UNSEAL.

Initialization. The protected variables of the TPM must be initialized before use. We write TPM_0 for the initialization command. Informally, this command runs once as the TPM is manufactured. It generates cryptographic keys and sets h_1, h_{17}, and c to zero. For cryptographic reasons, we also need to randomly sample $\mathcal{H}$ in a family of universal one-way hash functions; this is modelled as an implicit parameter ν for $\mathcal{H}$. Concretely, the public key of the TPM may also be certified by some authority, so that its high integrity can be dynamically verified.

$$\mathit{TPM}_0 \doteq \quad k^-_{\mathit{TPM}}, k^+_{\mathit{TPM}} := \mathcal{G}_e();\ s.ke := \mathcal{G}_{\mathcal{SE}}();\ s.ka := \mathcal{G}_{\mathcal{M}}();\ \nu := \mathcal{G}();\ h_1 := 0;\ h_{17} := 0;\ c := 0$$

5 CFLOW revisited

We describe the CFLOW compiler, giving its specification and outlining its algorithms; we refer to Fournet et al. (2009) for a detailed presentation. The compiler takes a source program plus security and locality policies, and outputs a cryptographically-protected distributed program. We improve on earlier work by handling more source programs, with endorsements and declassification, and by providing more precise theorems.

Source language, with locations. We consider a finite set of hosts, or locations, $\{1, 2, \ldots, i, b, c, v, \ldots, n\}$ intended to represent units of trust (principals) and of locality (runtime environments). The source language is the language of Section 2 extended with host annotations:

$$P ::= \ldots \mid b : P$$

The locality command $b : P$ states that command P should run on host b. Locality commands can be nested, as in $c : \{P_c; v : P_v\}$. We assume that every source program has a locality command at top level, setting an initial host. Since variables are transparently shared between hosts, locality annotations do not affect our semantics for commands.

Typing locality commands rules. We extend our security policy to assign a level $\Gamma(b)$ to each host b; this level indicates which variables b can read and write. We only consider robust hosts, such that $I(b) \leq R(C(b))$. We use the typing rule

$$\text{(TL{\scriptsize OCALITY})}\quad \frac{\vdash P : \ell \qquad I(b) \leq_I I(\ell)}{\vdash (b : P) : \ (\bot, I(\ell))}$$

The rule states that locality commands are public, thereby reflecting that the transfer of control between hosts can be observed by the adversary.

We illustrate CFLOW for the example in the introduction. The source code and its policy specify levels of protection, but leave the choice of cryptographic mechanisms to the compiler. The actual source and compiled programs are available online.

Example 4 (Applying for a loan: source code). The code is

$$b\text{: } \{x_b := e_b\};\ c\text{:}\{y_c := e_c\};\ v\text{: } \{x'_b, y'_c := f(x_b, y_c)\};\ b\text{: } \{\mathit{print}(x'_b)\};\ c\text{: } \{\mathit{print}(y'_c)\}$$

It involves three hosts: a client c with $\Gamma(c) = (C_c, I_c)$, a bank b with $\Gamma(b) = (C_b, I_b)$, and a 'virtual' host v with $\Gamma(v) = (C_c \sqcup C_b, I_c \sqcap I_b)$ for the TPM-attested code on the client machine. All variables indexed by b, c or v are private to b, c or v, respectively. For instance, $\Gamma(x_c) = (C_c, I_c)$. The bank and the client first write their secret values (in x_b and y_c); then v computes the two results (x'_b and y'_c); finally, the bank and the client print them (locally). With this source command, for instance, an adversary at the level of the bank cannot read the client secret, and vice versa.

Compiler transformation. The compiler inputs a command with localities P and a security policy Γ, and outputs an initialization command, Q_0, used to specify initial trust assumptions, plus a series of commands $\widetilde{Q}$ that include one command Q_i for each host i that occurs in the source program. We write Γ' for the security policy of these commands. Informally, Q_i is a single command that implements and schedules all code fragments of P located on i. After type checking, the compilation proceeds in 4 passes:

1. The source program is sliced into local threads, each running on a single host.

2. The distributed control flow between threads is protected, using dynamic checks on auxiliary program-counter variables, so that the adversary cannot run high-integrity threads out of schedule.
3. Relying on a single-static-assignment transformation, each variable shared between different hosts (including the program counters) is replaced by a series of local replicas, with explicit transfers between replicas.
4. Depending on their security levels, memory transfers between replicas are cryptographically protected, by inserting encryptions to transfer instead low-confidentiality encrypted values, and inserting authentication primitives to transfer instead low-integrity values. The compiler determines which symmetric keys to use for these operations, and generates an initial key-exchange protocol to distribute them. After this pass, the only variables shared between different hosts are (1) the signature verification keys used by the initial protocol, and (2) public-untrusted variables at level $(\perp, \top)$. (The compiler also generates untrusted code for scheduling these commands and transferring public-untrusted data.)

These 4 passes define a program transformation $Q_0, \widetilde{Q} \doteq [\![P]\!]$ such that each command Q_i has type ℓ_i^I. Next, we instantiate Definitions 3 and 4 for this transformation.

Source programs and their adversaries. We let source programs range over well-typed polynomial commands with locality annotations Informally, source programs enable their active adversaries to run whenever they pass control between hosts (since the adversary controls at least the network). To each source command P, we associate the command context $\widehat{P}$ obtained from P by replacing every subcommand of the form $b : P'$ by a command context with two command variables $X; P'; X'$. For a given adversary level α, we let $\widetilde{A}$ range over tuples of polynomial adversary commands, with one command for each command variable. Hence, $\widehat{P}[\widetilde{A}]$ ranges over commands that interleave the code of P with adversary commands. (This is analogous to models of non-interference for concurrency, where the adversary runs between any two program steps.) Thus, we define source evaluation by

$$\langle\!\langle P, A, \mu \rangle\!\rangle \doteq \rho_\infty \langle \widehat{P}[A], \mu \rangle$$

Implementations and their adversaries. Implementation programs range over our compiler outputs $Q_0, \widetilde{Q}$. Once Q_0 has run, we simulate concurrency by letting the adversary explicitly schedule commands $(\widetilde{Q})$ that represent parallel threads of computation (rather than having $\widehat{P}$ schedule $\widetilde{A}$). The resulting low-level model realistically accounts for all interleavings of these threads. Implementation adversaries range over adversaries command contexts A' with one hole for each host. Thus, we define target evaluation by

$$\langle\!\langle (Q_0, \widetilde{Q}), A', \mu' \rangle\!\rangle \doteq \rho_\infty \langle Q_0; A'[\widetilde{Q}], \mu' \rangle$$

We let π be the erasure of all variables added in Γ': $\pi(\mu') = \mu'_{|dom(\mu)}$ and we define equivalence on final memory distributions ($\rho_0 \approx \rho_1$) as computational indistinguishability: for all polynomial commands T, $|\Pr[\langle T, \rho_0 \rangle; g = 0] - \Pr[\langle T, \rho_1 \rangle, g = 0]|$ is negligible. (We use indistinguishability instead of distribution equality because our compiler relies on cryptographic security assumptions.)

With the definitions above, our new compilation theorem for CFLOW is

Theorem 5. *Let $\alpha \in \mathcal{L}$ be such that R is robust against α. Let Γ be a robust policy. $([\![\cdot]\!], \pi)$ is secure; and $([\![\cdot]\!], \pi)$ is correct when $\alpha = (\bot_C, \top_I)$.*

The theorem demands that the source policy Γ be robust (Definition 1), so that the adversary can read any shared variable that it can write. This hypothesis stems from our decision to support endorsements in source programs. In particular, the control flow integrity enforced by pass 2 may otherwise fail to protect programs that combine endorsements and declassifications, as illustrated below.

Example 5 (Non-robust shared variables). Consider a source program that writes a secret s with $\Gamma(s) = \mathit{HH}$ into a (non robust) variable x with $\Gamma(x) = \mathit{HL}$, then erases the content of x, and finally declassifies x by copying it to p with $\Gamma(p) = \mathit{LL}$:

$$P \doteq 1\!:\!\{x := s\}; 2\!:\!\{x := 0\}; 3\!:\!\{p := x\}$$

Let $\alpha = \mathit{LL}$. With our source semantics, the command context

$$\widehat{P} \doteq X_1; x := s; X_2; x := 0; X_3; p := x; X_4$$

ensures that p finally contains either 0 or a value written by the adversary, but not the value of s. In the implementation, however, the two local commands at hosts 1 and 2 have low integrity, so pass 2 does not guarantee their sequential execution, and an implementation adversary that schedules Q_2 before Q_1 lead Q_3 to declassify s into p.

Cryptographic protection (pass 4) is also problematic for programs that share non-robust variables, such as x in Example 5. Although their confidentiality is protected by encryption, an implementation adversary can swap their contents (by swapping their encrypted values) and similarly lead the program to declassify the wrong data.

Simulation vs non-interference. Instead of Theorem 5, Fournet et al. (2009) show that two classes of information-flow properties of the source program are preserved in the implementation. Our security result is more precise; it guarantees that, for any attack against our implementation, there is also an attack against the source program, with the same information leakage. The theorem below confirms that our new result generally subsumes theirs, and thus yields strong computational non-interference properties. (We refer to Fournet et al. (2009) and to the full paper for the definitions and discussion of their notion of computational non-interference for confidentiality and for integrity.)

Theorem 6 (Computational Non-Interference). *If a transformation $([\![\cdot]\!], \pi)$ is secure, then it preserves computational confidentiality and integrity.*

Example 6 (Simplified implementation). Continuing with our example, and in preparation for the next section, we give a simplified, hand-written implementation of Example 4 that illustrates the main mechanisms of CFLOW while avoiding those irrelevant here. For instance, the ordering of $x_b := e_b$ and $y_c := e_c$ is irrelevant; the ordering of $x_b := e_b$ and $x'_b, y'_c := f(x_b, y_c)$ is protected because $x'_b, y'_c := f(x_b, y_c)$ does not run unless x_b has been verified. So, instead of the globally shared and signed programs counter, we use one local anti-replay counter for each host. Communications between b and v are cryptographically protected, but we let v and c share local memory (since v will run on c's machine). Otherwise, all the new (communication) variables are public and untrusted; the only shared high-integrity variables are the public keys (k_b^+ and k_v^+). The commands are :

```
$Q_0 \doteq$  $k_b^-$,$k_b^+$ := $\mathcal{G}_e()$; $k_v^-$,$k_v^+$ := $\mathcal{G}_e()$
$Q_b \doteq$  if $c_b$=1 then { $c_b$++; $x_b$:= $e_b$; $x_e$ := $\mathcal{E}(x_b, k_v^+)$; $x_s$ := $\mathcal{S}(x_e, k_b^-)$ }
              else if $c_b = 2$ then { $c_b$++; if $\mathcal{V}(x'_e, x'_s, k_v^+)$ then $print(\mathcal{D}(x'_e, k_b^-))$ }
$Q_c \doteq$  if $c_c$=1 then { $c_c$++; $y_c$ := $e_c$ } else if $c_c$=2 then { $c_c$++; $print(y'_c)$ }
$Q_v \doteq$  if $c_v$=1 then
              { $c_v$++; if $\mathcal{V}(x_e, x_s, k_b^+)$ then { $x_v := \mathcal{D}(x_e, k_v^-)$; $x'_v, y'_c := f(x_v, y_c)$;
                                                   $x'_e$ := $\mathcal{E}(x'_v, k_b^+)$; $x'_s$ := $\mathcal{S}(x_e, k_v^-)$ } }
```

6 Implementing virtual hosts on TPMs

Section 5 shows how to compile an imperative program with shared access-controlled memory into a distributed program protected by cryptography. The resulting program runs on a series of machines, and preserves the security properties of the source program, subject to the assumption that each local command Q_b of the distributed implementation runs on a machine with (at least) the security of its declared level ℓ_b. However, for many useful programs, it is difficult to find such machines for the most trusted parts of the computation.

This section introduces a transformation that relies on secure instructions to boot trusted virtual machines. This transformation applies to any distributed programs, including those produced by CFLOW in Section 5. Before giving general definitions and theorems, we illustrate the transformation on Example 6.

Example 7 (Securely booting Q_v). The command Q_v requires a machine trusted by both the client and the bank. Assume that the bank trusts the client TPM for running Q_v and knows its public key for attestation. We may use the code

```
$Q_0 \doteq$  $k_b^-$,$k_b^+$ := $\mathcal{G}_e()$; $k_{TPM}^-$,$k_{TPM}^+$ := $\mathcal{G}_e()$; $c$ := 0;
$Q_b \doteq$  if $c_b$=1 then { $c_b$++; $x_b$:= $e_b$;
                          if VERIFY($\mathcal{H}(\langle K_v \rangle)$,$k_v^+$, $cert_v$)
                          [ $b.k_v^+ := k_v^+$; $x_e$ := $\mathcal{E}(x_b, k_v^+)$; $x_s$ := $\mathcal{S}(x_e, k_b^-)$ ] }
              else if $c_b$=2 then { $c_b$++; if $\mathcal{V}(x'_e, x'_s, k_v^+)$ then $print(\mathcal{D}(x'_e, k_b^-))$ }
$Q_c \doteq$  if $c_c$=1 then { $c_c$++; $y_c$ := $e_c$ } else if $c_c$=2 then { $print(y'_c)$ }
$Q_v \doteq$  $kernel$ := $\langle K_v \rangle$; SKINIT
$K_v \doteq$  if $c$=0 then
                { INC; $k_v^-$,$k_v^+$ := $\mathcal{G}_e()$; $cert_v$ := ATTEST($k_v^+$); $key$ := SEAL($k_v^-$,$h$) }
              else if $c$=1 then
                { INC; $k_v^-$ := UNSEAL($key$,$h$);
                  if $\mathcal{V}(x_e, x_s, k_b^+)$ then { $x_v := \mathcal{D}(x_e, k_v^-)$; $x'_v, y'_c := f(x_v, y_c)$;
                                                   $x'_e$ := $\mathcal{E}(x'_v, `k_b^+)$; $x'_s$ := $\mathcal{S}(x'_e, k_v^-)$ } }
```

In contrast with the host commands of Section 5, TPM-attested host commands do not have a persistent, protected local memory to keep their trusted key pair. Instead, as we dynamically set up host v, we generate its key pair, we use remote attestation to convince the bank to encrypt its secret towards the implementation of Q_v, and we simulate the persistent local memory using seal/unseal and the TPM counter to protect against replays.

A general transformation. Our transformation takes as input a policy $\mathit{\Gamma}$, a tuple of typed commands $\widetilde{Q}$ for hosts $\widetilde{b}$, including the command of a virtual host v, and a subset of the variables $\widetilde{x}$ of Q_v (informally, $\widetilde{x}$ represents the private, trusted local state of v). It assumes the existence of a TPM on host a, at least as trusted as v and not used in the source program. It generates an implementation policy $\mathit{\Gamma}'$ and a series of implementation commands Q'_0, $\widetilde{Q}'$, defined below. We let $\widetilde{Q}$ range over commands such that

1. no command in $\widetilde{Q} \setminus Q_v$ accesses $\widetilde{x}$ (i.e. the variables of $\widetilde{x}$ are local to Q_v);
2. Q_v does not read $\widetilde{x}$ before initializing them;
3. $\vdash Q_b : \ell_b^I$ for $b \in \widetilde{b}$;
4. $\vdash Q_v : \ell_{\mathit{TPM}}^I$ (i.e. the TPM is as trusted as Q_v); and
5. $R(C(\ell_{\mathit{TPM}})) \not\leq I(\ell_b)$ for $b \in \widetilde{b}$ (i.e. no host can access the TPM private variables).

Initialization. Recall that the commands $\widetilde{Q}$ (including Q_v) rely on trusted variables formally initialized in Q_0. In contrast, our implementation of Q_v uses SKINIT, so its own initialization is deferred until runtime and must be explicitly coded. For simplicity, we assume that, for each host $b \in \widetilde{b}$, initialization is a command of the form $Q_{0,b} \doteq k_b^-, k_b^+ := \mathcal{G}_e()$;... that writes private variables and generates a single keypair, with v initialized last. We also assume that the keys written in Q_0 are not overwritten in $\widetilde{Q}$. (These assumptions hold with the CFLOW compiler, up to a reordering of hosts.)

6. $Q_0 \doteq (Q_{0,b};)_{b \neq v}; Q_{0,v}$;
7. no command in $\widetilde{Q} \setminus Q_b$ accesses variables written in $Q_{0,b}$ except for k_b^+ for $b \in \widetilde{b}$ (i.e. the variables initialized in $Q_{0,b}$ are local to Q_b);
8. no command in $\widetilde{Q}$ writes k_b^+, k_b^- for $b \in \widetilde{b}$.

Implementation of Q_0. Initialization is obtained from Q_0 by adding initialization for the TPM and removing initialization for v: $Q'_0 \doteq \mathit{TPM}_0$; $(Q_{0,b};b.k_v^+ := 0;)_{b \neq v}$.

Implementation of Q_v. The command Q'_v uses SKINIT to dynamically launch a secure kernel K_v that implements Q_v:

```
$Q'_v \doteq$ $\mathit{kernel} := \langle K_v \rangle$; $a$.SKINIT
$K_v \doteq$ $a$.INC;
        if $a.c = c_v^0$+1 then { $v.c := a.c$; $Q_{v,0}$; $\mathit{cert}_v := a.\mathrm{ATTEST}_{17}(k_v^+)$}
                      else { $k_v^- | k_v^+ | v.c | v.\widetilde{x} := a.\mathrm{UNSEAL}_{17}(\mathit{store}_v, a.h_{17})$;
                            if $v.c = a.c$ then $Q_v$ $\{v.x/x, x \in \widetilde{x}\}\{`k_b^+/k_b^+, b \neq v\}\}$;
        $\mathit{store}_v := a.\mathrm{SEAL}_{17}((k_v^-, k_v^+ | v.c\text{+}1 | v.\widetilde{x}), a.h_{17}))$; $k_v^-, v.c, v.\widetilde{x} := 0$
```

The variables $v.\widetilde{x}$ (and $v.c$) are volatile for each run of Q'_v; they can be public and untrusted, but must be cleared before returning. By eliminating trusted and confidential variables, the transformation lowers the level of Q_v hence the level required to run it.

The identity of K_v is verified by the other hosts, so the command Q'_v itself need not be trusted. The other new variables k_v^+, cert$_v$, store$_v$ are also (formally) public and untrusted. They need not be trusted for security, although an adversary that can overwrite them may cause the system to fail. The value c_v^0 is a constant of the program and corresponds to the initial value of the TPM monotonic counter.

Implementation of Q_b for $b \neq v$, $b \in \widetilde{b}$. As it runs for the first time, each host command Q'_b verifies the attested public key for v and stores it in a new local variable, $b.k_v^+$, a local trusted copy to be used instead of k_v^+ in Q_b. To this end, Q'_b recomputes the expected value of h, including the values for c_v^0 and any other keys $k_{b'}^+$ used in Q_v, and verifies its concatenation with the received public key k_v^+ using the trusted verification key of the supporting TPM.

$$\begin{aligned} Q'_{b,i} \doteq \text{ if } b.k_v^+ = 0 \text{ then } \{ & \text{ if } a.\text{VERIFY}(v.\mathcal{H}(\langle K_v \rangle), k_v^+, cert_v)[\ b.k_v^+ := k_v^+]\ \} \\ & \text{else } \{\ Q_{b,i}\{b.k_v^+/k_v^+\}\ \} \end{aligned}$$

Implementation of Γ. The formal implementation of Γ is $\Gamma' = \Gamma\{v.\widetilde{x}, v.c, k_v^+, cert_v, store_v \mapsto (\bot, \top)\}\{b.k_v^+ \mapsto \ell_b, b \neq v\}$.

Security and functional correctness. We express the security and functional correctness of our transformation as instances of definitions 3 and 4. Source programs range over commands $\widetilde{Q}$ that meet conditions 1–8 above. Source adversaries are parameterized by α and range over adversary command contexts for Γ. The commands of the distributed programs are scheduled by the adversary after the execution of Q_0; formally:

$$\langle\!\langle \widetilde{Q}, A, \mu \rangle\!\rangle \doteq \rho_\infty \langle Q_0; A[\widetilde{Q}], \mu \rangle$$

Implementation programs $[\![\widetilde{Q}]\!]$ range over $\widetilde{Q}'$, as defined above. Implementation adversaries are parameterized by α and range over valid adversary command contexts for Γ'. The commands of the distributed program are scheduled by the adversary after the execution of Q'_0 but additionally, the adversary and Q_v have access to protected versions of the subroutine SKINIT$_a$: we let

$$\begin{aligned} S' &\doteq Q'_0; A'[a.\text{SKINIT}_{\alpha^I}, \widetilde{Q}', Q'_v[a.\text{SKINIT}_{\ell_v'^I}]] \\ \langle\!\langle \widetilde{Q}', A', \mu' \rangle\!\rangle' &\doteq \rho_\infty(\langle S', \mu' \rangle) \end{aligned}$$

where $a.$SKINIT$_\ell$ runs $a.$SKINIT after testing that *kernel* contains code typed at level ℓ.

We let π be the erasure of all variables added in Γ': $\pi(\mu') = \mu'_{|dom(\mu)}$ and we define equivalence on final memory distributions ($\rho_0 \approx \rho_1$) as computational indistinguishability: for all polynomial commands T such that T does not read $\{\widetilde{x}, k_v^-, k_v^+\}$, $|\Pr[\langle T, \rho_0 \rangle; g = 0] - \Pr[\langle T, \rho_1 \rangle; g = 0]|$ is negligible. Relying on these definitions, our main theorem for virtual hosts on TPMs is

Theorem 7. *Let $\alpha \in \mathcal{L}$ be such that R is robust against α. Let Γ be a robust policy. $([\![\cdot]\!], \pi)$ is secure; $([\![\cdot]\!], \pi)$ is correct when $\alpha = (\bot_C, \top_I)$.*

Since its input and output are in the same format, the transformation and its theorem can be applied several times to implement a series of virtual hosts using different TPMs.

Acknowledgments. We thank Karthik Bhargavan, Gurvan le Guernic, Jean-Jacques Lévy, Himanshu Raj, Tamara Rezk, and the anonymous reviewers for their help.

References

M. Abadi and T. Wobber. A logical account of NGSCB. *In FORTE 2004*, pages 1–12, 2004.

AMD. AMD64 virtualization: Secure virtual machine architecture reference manual.

A. Askarov and A. Myers. A semantic framework for declassification and endorsement. *Prog. Languages and Systems*, pages 64–84, 2010.

A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *CSF*, 2009.

S. Balfe and K. G. Paterson. e-EMV: Emulating EMV for internet payments using trusted computing technology. *STC 2008*, pages 81–92, 2008.

S. Chong and A. C. Myers. Decentralized robustness. *19th IEEE CSFW, 2006*, page 12, 2006.

S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. *CACM*, 2009.

A. Datta, J. Franklin, D. Garg, and D. Kaynar. A logic of secure systems and its application to trusted computing. In *S&P'09*, 221–236, 2009.

D. E. Denning. A lattice model of secure information flow. In *CACM*, 1976.

W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE TIT*, 1976.

C. Fournet and T. Rezk. Cryptographically sound implementations for typed information-flow security. In *POPL'08*, pages 323–335. Jan. 2008.

C. Fournet, G. Le Guernic, and T. Rezk. A security-preserving compiler for distributed programs: from information-flow policies to cryptographic mechanisms. In *CCS'09*, ACM, 2009.

D. Grawrock. TCG Specification Architecture Overview, Rev. 1.4, 2007.

S. Gürgens, C. Rudolph, D. Scheuermann, M. Atts, and R. Plaga. Security evaluation of scenarios based on the TCGs TPM specification. *Computer Security–ESORICS 2007*.

J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten. Lest we remember: cold-boot attacks on encryption keys. *CACM* 2009.

Intel. Intel Trusted Execution Technology Software Development Guide, 2009.

J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *3rd ACM SIGOPS/EuroSys*, pages 315–328. ACM, 2008.

J. McCune, N. Qu, Y. Li, A. Datta, V. Gligor, and A. Perrig. Efficient TCB Reduction and Attestation. *CMU-CyLab-09-003*, 9, 2009.

Microsoft. Windows BitLocker drive encryption, 2006.

J. Millen, J. Guttman, J. Ramsdell, J. Sheehy, B. Sniffen, and M. Bedford. Analysis of a measured launch, MITRE, 2007.

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. *TOSEM* 2000.

A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow, 2001.

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

F. Pottier and V. Simonet. Information flow inference for ML. *ACM TOPLAS*, 2003.

A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J-SAC*, 2003.

A. Sabelfeld and A. C. Myers. A model for delimited information release. *Software Security-Theories and Systems*, pages 174–191, 2004.

Trusted Computing Group. Client Specific TPM Interface Specification (TIS), Version 1.2., 2005.

Trusted Computing Group. TCG Software Stack (TSS 1.2). Trusted Computing Group, 2006.

S. Zdancewic and A. C. Myers. Robust declassification. In *CSFW'01*, pages 15–23, 2001.

S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *TOCS* 2002.

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