Provably secure compilation of side-channel countermeasures

Gilles Barthe*, Benjamin Grégoire†, Vincent Laporte*
*IMDEA Software Institute, Madrid, Spain
gilles.barthe@imdea.org vlaporte@imdea.org
†Inria, Sophia-Antipolis, France
benjamin.gregoire@inria.fr

Abstract

Software-based countermeasures provide effective mitigation against side-channel attacks, often with minimal efficiency and deployment overheads. Their effectiveness is often amenable to rigorous analysis: specifically, several popular countermeasures can be formalized as information flow policies, and correct implementation of the countermeasures can be verified with state-of-the-art analysis and verification techniques. However, in absence of further justification, the guarantees only hold for the language (source, target, or intermediate representation) on which the analysis is performed.

We consider the problem of preserving side-channel countermeasures by compilation, and present a general method for proving that compilation preserves software-based side-channel countermeasures. The crux of our method is the notion of 2-simulation, which adapts to our setting the notion of simulation from compiler verification. Using the Coq proof assistant, we verify the correctness of our method and of several representative instantiations.

1. Introduction

Side-channel attacks are physical attacks in which malicious parties extract confidential and otherwise protected data from observing the physical behavior of systems. Side-channel attacks are also among the most effective attack vectors against cryptographic implementations, as witnessed by an impressive stream of side-channel attacks against prominent cryptographic libraries. Many of these attacks fall under the general class of timing-based attacks, i.e. they exploit the execution time of programs. In their simplest form, timing-based side-channel attacks only use very elementary facts about execution time [16], [11]: for instance, branching on secrets may leak confidential information, as the two branches may have different execution times. Further instances of these attacks include [2], [1]. However, advanced forms of timing-based side-channel attacks also exploit facts about the underlying architecture. Notably, cache-based timing attacks exploit the latency between cache hits and cache misses. Cache-based timing attacks have been used repeatedly to retrieve almost instantly cryptographic keys from implementations of AES and other libraries—see for example [9], [23], [27], [14], [18], [28], [13]. Similarly, data timing channels exploit the timing variability of specific operations—i.e. operations whose execution time depends on their arguments [19].

Numerous countermeasures have been developed in response to these attacks. Hardware-based countermeasures propose solutions based on modifications of the micro-architecture, e.g. providing hardware support for AES instructions, or making caches security-sensitive. These countermeasures are effective, but their deployment may be problematic. In contrast, software-based countermeasures propose solutions that can be implemented at language level, including secure programming guidelines and program transformations which automatically enforce these guidelines. Popular software-based

countermeasures against timing-based side-channel attacks include the program counter [20] and the constant-time [9] policies. The former requires that the control-flow of programs does not depend on secrets, and provides effective protection against attacks that exploit secret-dependent control-flow, whereas the latter additionally requires that the sequence of memory accesses does not depend on secrets, and provides an effective protection against cache-based timing attacks.

Software-based countermeasures are easy to deploy; furthermore, they can be supported by rigorous enforcement methods. Specifically, the prevailing approach for software-based countermeasures is to give a formal definition, often in the form of an information flow policy w.r.t. an instrumented semantics that records information leakage. Broadly speaking, the policies state that two executions started in related states (from an attacker's point of view) yield equivalent leakage. These policies can then be verified formally using type systems, static analyses, or SMT-based methods. Formally verifying that software-based countermeasures are correctly implemented is particularly important, given the ease of introducing subtle bugs, even for expeert programmers.

However, and perhaps surprisingly, very little work has considered the problem of carrying software-based countermeasures along the compilation chain. As a result, developers of cryptographic libraries are faced with the following dilemma whenever implementing a software-based countermeasure:

- implement and verify their countermeasure at target level, with significant productivity costs, because of the complexity to reason about low-level code;
- implement and verify their countermeasure at source-level and trust the compiler to preserve the countermeasure.

Sadly, none of the options is satisfactory. In particular, it must be noted that compilers do not generally preserve security policies. Here we list a few illustrative examples. Kaufmann and coauthors [15] build a timing attack against an implementation of the scalar product on an elliptic curve that is constant-time (assuming that the 64-bit multiplication does not leak information about its operands). Indeed, in that setting, the compiler optimizes the multiplications so that they run faster on small values, resulting in an information leak. Comparison of fixed-length bitstrings is often seen as a simple atomic step at high-level; however, it can be compiled to a loop with early exit. This common optimization may reveal the position of first different bit, hence break the constant-time property of the source program. Lazy boolean operators can be seen as secure (i.e., they do not leak any information) at high-level. Even if they are not used in conditional instructions, hey are often compiled using such leaky branches, breaking constant-time. Memoization is a general program transformation that may be applied at compile-time [26]. It replaces computations with table look-ups, introducing memory accesses that may depend on sensitive data. Such transformations may indeed break constant-time. A common technique to write constant-time implementation is to used a 1-bit multiplication (by a sensitive bit) instead of a conditional branch. However, powerful value analyses used in optimizing compilers may recognize that the multiplicand is only one bit and turn the constant-time operation into an branch, defeating the point of constant-time programming.

To address this problem, we propose the first general framework for proving that software countermeasures are preserved by compilation. As a starting point, we define observational non-interference, which formalize a broad class of software countermeasures, including the program counter and constant-time policies. Next, we give sufficient conditions for preservation of observational non-interference by compilation. Our conditions are based on an adaptation of the usual notions of simulation from compiler verification; we call our notions 2-simulations, since they seem applicable to the broader context of 2-safety properties—of which observational non-interference policies are an instance. As for

simulations, 2-simulations come in several flavours (lockstep, manysteps, and general); each of them establishes preservation of constant-time. Crucially, preservation proofs are modular: compiler correctness is assumed, and does not need to be re-established. This allows for a neat separation of concerns and incremental proofs (e.g. first prove compiler correctness, then preservation of constant-time), and eases future applications of our method to existing verified compilers. We prove the correctness of our framework and demonstrate its usefulness by deriving preservation of observational non-interference for a representative set of compiler optimizations. Our proofs are formally verified using the Coq proof assistant, yielding the first verified middle-end compiler that preserves the constant-time policy.[1] Furthermore, we discuss several generalizations of our method and discuss its applications to existing realistic compilers.

Overall, our framework lays previously missing theoretical foundations for preservation of software-based side-channel countermeasures by compilation, and provides a clear methodology for developing a next generation of secure compilers that go beyond functional correctness.

Summary of contributions. The main technical contributions of the paper include:

- we provide the first methodology to prove preservation of software countermeasures by compilation;
- we study common classes of compilation passes and prove that they preserve observational non-interference;
- we provide mechanized proofs of correctness of our method, and of the instantiations to specific optimizations.

2. Problem statement

2.1. Definitions

Observational policies are a variant of information flow policies for programs that carry an explicit notion of leakage.

We model leakage using a monoid $(\mathcal{L}, \cdot, \epsilon)$. We model the behavior of programs using an instrumented operational semantics. Formally, we assume given a set $\mathcal{S}$ of states with a distinguished subset $\mathcal{S}_f$ of final states, and assume that the behaviors of programs is given by labelled transitions of the form $a \xrightarrow{t} a'$, where a and a' are states and t is the leakage associated to the one step execution from a to a'. We write $a \to a'$ when t is irrelevant, and require that $a \to a'$ implies $a \notin \mathcal{S}_f$. The converse may fail, i.e. $a \notin \mathcal{S}_f$ does not imply the existence of a state a' such that $a \to a'$. We say that a state a is safe, written $\mathsf{safe}(a)$, iff $a \in \mathcal{S}_f$ or there exists $a \in \mathcal{S}$ such that $a \to a'$.

For simplicity, we assume that the semantics is deterministic, i.e., for all $a, a_1, a_2 \in \mathcal{S}$ and $t_1, t_2 \in \mathcal{L}$,

$$(a \xrightarrow{t_1} a_1 \wedge a \xrightarrow{t_2} a_2) \implies (a_1 = a_2 \wedge t_1 = t_2)$$

Multi-step execution $a \xrightarrow{t}{}^{+} a'$ is defined by the clauses:

$$\frac{a \xrightarrow{t} a'}{a \xrightarrow{t}{}^{+} a'} \qquad\qquad \frac{a \xrightarrow{t} a' \qquad a' \xrightarrow{t'}{}^{+} a''}{a \xrightarrow{t \cdot t'}{}^{+} a''}$$

1. The whole development is available at https://sites.google.com/view/ctpreservation

n-step execution is defined similarly:

$$\frac{a \xrightarrow{t} a'}{a \xrightarrow{t}^1 a'} \qquad\qquad \frac{a \xrightarrow{t} a' \qquad a' \xrightarrow{t'}^n a''}{a \xrightarrow{t \cdot t'}^{n+1} a''}$$

We write $a \Downarrow_t$ iff there exists a state $a' \in \mathcal{S}_f$ such that $a \xrightarrow{t}^+ a'$. Formally, we assume a type $\mathcal{I}$ of input parameters and we see a program S as a function mapping input parameters to initial states. Therefore the set of initial state of the program S is $S(\mathcal{I})$. We assume that the type $\mathcal{I}$ is shared by all languages involved in the compilation chain. We say that a program S is safe iff for every $a \in S(\mathcal{I})$, a is safe and for every $a' \in \mathcal{S}$ such that $a \xrightarrow{t}^+ a'$, a' is safe.

Policies consist of two relations: a relation on input parameters, called the precondition, and a relation on leakage, called the postcondition.

Definition 1 (Policy)**.** An observation policy is given by a pair (ϕ, ψ) where $\phi \subseteq \mathcal{I} \times \mathcal{I}$ and $\psi \subseteq \mathcal{L} \times \mathcal{L}$.

Given two states $a,\ a' \in S(\mathcal{I})$, we write $\phi\ a\ a'$ if there exist $i,\ i' \in \mathcal{I}$ s.t. $S(i) = a \wedge S(i') = a' \wedge \phi\ i\ i'$.

We list several observation policies (all but the last one have been considered in the literature), and succinctly describe for each of them the leakage model and the definition of ψ—for now, we leave ϕ unspecified.

- program counter policy: control flow is leaked. Leakage is a list of boolean values containing all guards evaluated during this execution. ψ is equality;
- memory obliviouness: memory accesses are leaked. Leakage is a list of memory addresses accessed during this execution (but not their value). ψ is equality;
- constant-time policy: control flow and memory addresses are leaked. It combines the program counter and memory obliviousness policies. Leakage is an heterogeneous list of booleans and addresses. ψ is equality;
- step-counting policy: the number of execution steps is leaked. ψ is equality;
- resource-based non-interference: execution cost is leaked. Leakage is a single number, or a list of numbers (one number per instruction), depending on the granularity of the model. ψ is equality;
- size-respecting policy: size of operands is leaked for specific operators, e.g. division. Leakage is a list of sizes (taken from a finite set size. ψ is equality;
- constant-time policy with size: control flow, memory addresses, and size of operands are leaked. Leakage is an heterogeneous list of booleans, addresses, and sizes. ψ is equality;
- bounded difference policy: execution cost is leaked (as in cost obliviousness). ψ requires that the difference between the leakage at each individual step (or the accumulated leakage) does not exceed a given upper bound.

We next define two notions of satisfaction for policies. The first notion is termination-insensitive, and only considers terminating executions.

Definition 2 (Termination-insensitive observational non-interference)**.** A program S satisfies termination-insensitive observational non-interference (TIONI) w.r.t. policy (ϕ, ψ), written $S \models (\phi, \psi)$, iff for every states $a, a' \in S(\mathcal{I})$,

$$a \Downarrow_t \wedge\, a' \Downarrow_{t'} \wedge\, \phi\ a\ a' \implies \psi\ t\ t'$$

Moreover, we say that S is TI-constant-leakage w.r.t. ϕ iff $P \models (\phi, =)$.

The second notion is termination-sensitive and compares executions of equal length.

Definition 3 (Termination-sensitive observational non-interference)**.** A program S satisfies termination-sensitive observational non-interference (TSONI) w.r.t. policy (ϕ, ψ), written $S \models (\phi, \psi)$, iff for every states $a, a' \in S(\mathcal{I})$, and $b, b' \in \mathsf{S}$ and $n \in \mathbf{N}$,

$$a \xrightarrow{t}{}^{n} b \wedge a' \xrightarrow{t'}{}^{n} b' \wedge \phi\ a\ a' \implies \psi\ t\ t'$$

Moreover, we say that S is TS-constant-leakage w.r.t. ϕ iff $P \models (\phi, =)$.

The termination-sensitive definition is not meaningful for all policies. In particular, all programs are TSONI for the step-counting policy. However, the following lemma shows that TSONI is stronger than TIONI under certain conditions.

Definition 4 ((ϕ, ψ)-equiterminating)**.** (ϕ, ψ) are equiterminating if for all program S and states $a, a' \in S(\mathcal{I})$ such that $\phi\ a\ a'$ and for every executions $a \xrightarrow{t}{}^{n} b$ and $a' \xrightarrow{t'}{}^{n'} b'$ such that the traces satisfy the policy $\psi\ t\ t'$, then either both or none of the end states are final ($b \in \mathcal{S}_f \iff b' \in \mathcal{S}_f$).

Many policies, such as those of Figure 4, are equiterminating.

Lemma 1. *If S is TSONI w.r.t. policy (ϕ, ψ) and (ϕ, ψ) are equiterminating then it is also TIONI w.r.t. policy (ϕ, ψ).*

Proof. Let $a, a' \in S(\mathcal{I})$ s.t. $\phi\ a\ a'$. Assume $a \xrightarrow{t}{}^{n} b$ and $a' \xrightarrow{t'}{}^{n'} b'$ and $b, b' \in \mathcal{S}_f$. WLOG, assume that $n \leq n'$. There exist $t_1,\ t_2,\ b_1$ such that $a' \xrightarrow{t_1}{}^{n} b_1 \xrightarrow{t_2}{}^{n'-n} b'$ and $t' = t_1 \cdot t_2$. Since S is TSONI we have $\psi\ t\ t_1$. Since $b \in \mathcal{S}_f$ and (ϕ, ψ) are equiterminating $b_1 \in \mathcal{S}_f$ and $b_1 = b'$ and $t' = t_1$ (i.e. $t_2 = \epsilon$), which implies TIONI. □

Remark. In the previous lemma, the hypothesis that (ϕ, ψ) are equiterminating can be replaced by TIONI w.r.t. ϕ and the step-counting policy. This allows to conclude that $n = n'$ and to finish the proof.

The notion of satisfaction naturally entails an order on policies. Specifically, we say that policy (ϕ, ψ) implies policy (ϕ', ψ'), written $(\phi, \psi) \Rightarrow (\phi', \psi')$, iff every program that satisfies TIONI w.r.t. (ϕ, ψ) also satisfies TIONI w.r.t. (ϕ', ψ')—alternatively, one can base the definition on TSONI. The following intuitive implications are true:

constant-time	$\Rightarrow$	program counter
constant-time	$\Rightarrow$	memory obliviousness
program counter	$\Rightarrow$	step counting

In addition, the following lemma proves a standard monotonocity property for observational non-interference.

Lemma 2 (Monotonicity of observational non-interference)**.** *If $\phi' \subseteq \phi$ and $\psi \subseteq \psi'$, then $(\phi, \psi) \Rightarrow (\phi', \psi')$.*

The lemma is often used implicitly for justifying the soundness of program analyses which prove observational non-interference w.r.t. $(\top, \psi)$ rather than observational non-interference w.r.t. (ϕ, ψ)—where $\top$ denotes the full relation.

2.2. Examples

We now make the definition of observation policies precise for a minimal imperative language shown in Figure 1. The type of input parameters is the set of environments. The language features a loop

$$\begin{array}{rcl} e & ::= & x \mid n \mid e\ o\ e \mid a[e] \\ c & ::= & \mathsf{skip} \mid x = e \mid a[e] = e \mid c; c \mid \mathsf{if}\ e\ c\ c \mid \mathsf{loop}\ c\ e\ c \end{array}$$

where x ranges over variables, a ranges over arrays and e ranges over expressions. All arrays a have a fixed size $|a|$.

Figure 1: Minimal language

construct of the form $\mathsf{loop}\ c_1\ e\ c_2$. The additional generality of the construct (over while loops) mildly simplifies the presentation of some optimizations. For clarity of exposition, only constant or binary operators are considered for building expressions.

Environments and semantics of expressions. An environment ρ is a pair (ρ_v, ρ_a), where ρ_v is a partial map from the set $\mathcal{X}$ of variables to integers, and ρ_a is a partial map from $\mathcal{A} \times \mathbf{Z}$, where $\mathcal{A}$ is the set of arrays, to integers, i.e. $\rho_v : \mathcal{X} \rightharpoonup \mathbf{Z}$ and $\rho_a : \mathcal{A} \times \mathbf{Z} \rightharpoonup \mathbf{Z}$.

Constants are interpreted as integers. The interpretation of a binary operator o is given by a pair of functions $(\overline{o}, \underline{o})$ of type $\mathbf{Z} \times \mathbf{Z} \to \mathbf{Z}$ and $\mathbf{Z} \times \mathbf{Z} \to \mathcal{L}$, modelling functional behavior and leakage respectively. We allow the first function to be partial. As usual, we model partial functions using a distinguished value $\perp$ for undefined and assume that errors propagate.

The interpretation $[e]_\rho$ and leakage $\mathsf{leak}(e, \rho)$ of an expression e in an environment ρ are elements of $\mathbf{Z}$ and $\mathcal{L}$ respectively. Figure 2 defines the functions formally. The evaluation of variables and constants generates no leakage; for operators, the leakage corresponds to the leakage of subexpressions and the specific leakage of the operation $\underline{o}\ [e_1]_\rho\ [e_2]_\rho$. Note that, as for other operators, the definition of leakage for array access is parametrized by a leakage function $\lambda_{\mathsf{a}} : \mathcal{A} \times \mathbf{Z} \to \mathcal{L}$.

States and semantics of commands. States are pairs of the form $\{c, \rho\}$ where c is a command and ρ is an environment. We use $s.\mathsf{cmd}$ and $s.\mathsf{env}$ to denote the first and second component of a state. The instrumented semantics of commands is modelled by statements of the form $\{c, \rho\} \xrightarrow{t} \{c', \rho'\}$, and is parametrized by constants $\lambda_{\mathsf{skip}}, \lambda_{\mathsf{loop}} \in \mathcal{L}$, functions $\lambda_{\mathsf{v}} : \mathcal{X} \times \mathbf{Z} \to \mathcal{L}$, and $\lambda_{\mathsf{if}} : \mathbf{Z} \to \mathcal{L}$. We use the standard notation $\cdot\{\cdot \leftarrow \cdot\}$ for updating environments.

The semantics is standard, except for the semantics of loops, and the definition of the leakage. The loop command $\mathsf{loop}\ c_1\ e\ c_2$ first executes c_1 (unconditionally, as the *do-while* command), then evaluates e: if e evaluates to true (i.e. $[e]_\rho \neq 0$) the command c_2 is executed and the loop command is evaluated w.r.t. the updated environment (as in the *while-do*), else if e evaluates to false (i.e. $[e]_\rho = 0$) the command terminates immediately.

The leakage of a command is the leakage of the expressions evaluated during the execution of the command plus a specific leakage. For array assignment, the specific leakage depends of the *address* $\lambda_{\mathsf{a}}(a, [e_1]_\rho)$. For conditionnal, the leakage depends of the branch taken.

The following lemma establishes that the instrumented semantics is deterministic, as required by our setting.

Lemma 3. *For all states a, b, b', if $a \xrightarrow{t} b$ and $a \xrightarrow{t'} b'$, then $b = b'$ and $t = t'$.*

Leakage models. Figure 3 instantiates the functions for different leakage models. We use the notation $[a]$ to represent the list with a single element a.

$$[n]_\rho = n \qquad [x]_\rho = \rho_v(x) \qquad\qquad \mathsf{leak}(n,\rho) = \epsilon \qquad \mathsf{leak}(x,\rho) = \epsilon$$
$$[a[e]]_\rho = \rho_a(a, [e]_\rho) \qquad\qquad \mathsf{leak}(a[e],\rho) = \mathsf{leak}(e,\rho)\cdot\lambda_{\mathsf{a}}(a,[e]_\rho)$$
$$[e_1\ o\ e_2]_\rho = \overline{o}\ [e_1]_\rho\ [e_2]_\rho \qquad\qquad \mathsf{leak}(e_1\ o\ e_2,\rho) = \mathsf{leak}(e_1,\rho)\cdot\mathsf{leak}(e_2,\rho)\cdot \underline{o}\ [e_1]_\rho\ [e_2]_\rho$$

$$\begin{array}{rcll}
\{x = e, \rho\} & \xrightarrow{\mathsf{leak}(e,\rho)\cdot\lambda_{\mathsf{v}}(x,[e]_\rho)} & \{\mathsf{skip}, \rho\{x \leftarrow [e]_\rho\}\} & \\
\{a[e_1] = e_2, \rho\} & \xrightarrow{\mathsf{leak}(e_1,\rho)\cdot\mathsf{leak}(e_2,\rho)\cdot\lambda_{\mathsf{a}}(a,[e_1]_\rho)} & \{\mathsf{skip}, \rho\{a[[e_1]_\rho] \leftarrow [e_2]_\rho\}\} & \text{if } [e_1]_\rho < |a| \\
\{\mathsf{if}\ e\ c_1\ c_2, \rho\} & \xrightarrow{\mathsf{leak}(e,\rho)\cdot\lambda_{\mathsf{if}}([e]_\rho)} & \{c_1, \rho\} & \text{if } [e]_\rho \neq 0 \\
\{\mathsf{if}\ e\ c_1\ c_2, \rho\} & \xrightarrow{\mathsf{leak}(e,\rho)\cdot\lambda_{\mathsf{if}}([e]_\rho)} & \{c_2, \rho\} & \text{if } [e]_\rho = 0 \\
\{\mathsf{skip}; c_2, \rho\} & \xrightarrow{\lambda_{\mathsf{skip}}} & \{c_2, \rho\} & \\
\{\mathsf{loop}\ c_1\ e\ c_2, \rho\} & \xrightarrow{\lambda_{\mathsf{loop}}} & \{c_1; \mathsf{if}\ e\ (c_2; c)\ \mathsf{skip}, \rho\} & \text{where } c = \mathsf{loop}\ c_1\ e\ c_2
\end{array}$$

$$\frac{\{c_1, \rho\} \xrightarrow{t} \{c_1', \rho'\}}{\{c_1; c_2, \rho\} \xrightarrow{t} \{c_1'; c_2, \rho'\}}$$

Figure 2: Instrumented semantics of the while language

Instantiating the general setting. In order to complete the instantiation of the general setting, we must additionally provide the definition of $S(\mathcal{I})$ and $final$. These are respectively defined as the sets of states of the form $\{S, \rho\}$ and $\{\mathsf{skip}, \rho\}$, where ρ ranges over environments. Under these definitions, we can prove the aforementioned implications between the different policies.

Policies. Policies considered in the literature are generally of the form $(\doteq, =)$, where $\doteq$ represents low equivalence, and is defined relative to a security lattice and a security environment. We consider a lattice with two security levels: H, or high, for secret and L, or low, for public. Then, a security environment is a mapping from variables and arrays to security levels. Finally, two environments ρ and ρ' are low equivalent w.r.t. a security environment Γ iff they map public variables and arrays to the same values, i.e. $\rho_v(x) = \rho'_v(x)$ for all variables x such that $\Gamma(x) = L$ and $\rho_a(a, i) = \rho'_a(a, i)$ for all arrays a such that $\Gamma(a) = L$ and every $i \in \mathbf{N}$.

2.3. Secure compilation

The problem addressed in this paper is an instance of secure compilation. It can be stated as follows: given source and target languages (with potentially different leakage monoids), a compiler $[\![\cdot]\!]$ from source to target programs, and two policies (ϕ, ψ) and (ϕ, ψ'):

$$\forall S.\ S \models (\phi, \psi) \land \mathsf{safe}(S) \overset{?}{\Longrightarrow} [\![S]\!] \models (\phi, \psi')$$

A naive strategy for proving the implication would be to show that leakage is preserved by compilation, i.e. source executions with leakage t is compiled to target executions with equal leakage t. However, this strategy fails for most optimizations—and moreover, source and target languages do not even need to

Step-counting and cost obliviousness:
(for step-counting expressions have zero cost $\mathrm{leak}(e, \rho) = 0$)
$\mathcal{L} \triangleq \mathbf{N}$, $\cdot \triangleq +$, $\epsilon \triangleq 0$

$$\lambda_{\mathsf{skip}} = \lambda_{\mathsf{loop}} = \lambda_{\mathsf{a}}(a, z) = \lambda_{\mathsf{v}}(v, z) = \lambda_{\mathsf{if}}(z) = 1$$

Program counter:
$\mathcal{L} \triangleq \mathsf{list}(\mathbf{B})$, $\cdot \triangleq ++$, $\epsilon \triangleq \mathsf{nil}$

$$\lambda_{\mathsf{skip}} = \lambda_{\mathsf{loop}} = \lambda_{\mathsf{a}}(a, z) = \lambda_{\mathsf{v}}(x, z) = \epsilon \quad \lambda_{\mathsf{if}}(z) = [z \neq 0]$$

Memory obliviousness:
$\mathcal{L} \triangleq \mathsf{list}(\mathcal{X} + (\mathcal{A} \times \mathbf{N}))$, $\cdot \triangleq ++$, $\epsilon \triangleq \mathsf{nil}$

$$\lambda_{\mathsf{skip}} = \lambda_{\mathsf{loop}} = \lambda_{\mathsf{if}}(z) = \epsilon$$
$$\lambda_{\mathsf{a}}(a, z) = [(a, z)] \qquad \lambda_{\mathsf{v}}(x, z) = [x]$$

Constant-time:
$\mathcal{L} \triangleq \mathsf{list}(\mathbf{B} + (\mathcal{A} \times \mathbf{Z}))$, $\cdot \triangleq ++$, $\epsilon \triangleq \mathsf{nil}$

$$\lambda_{\mathsf{skip}} = \lambda_{\mathsf{loop}} = \epsilon \qquad \lambda_{\mathsf{if}}(z) = [z \neq 0]$$
$$\lambda_{\mathsf{a}}(a, z) = [(a, z)] \qquad \lambda_{\mathsf{v}}(x, z) = [x]$$

Size non-interference: (Only expressions leak)
$\mathcal{L} \triangleq \mathsf{list}(\mathsf{size})$, $\cdot \triangleq ++$, $\epsilon \triangleq \mathsf{nil}$

$$\lambda_{\mathsf{skip}} = \lambda_{\mathsf{loop}} = \lambda_{\mathsf{a}}(a, z) = \lambda_{\mathsf{v}}(x, z) = \lambda_{\mathsf{if}}(z) = \epsilon$$

Figure 3: Leakage models and observation policies

support the same notion of leakage. The failure of the naive strategy forces us to consider an alternative strategy based on 2-simulations, which we develop in the following sections.

Since the set of input parameters is shared among all the compilation phases, we also assume that we have a precondition ϕ which is shared by all the phases.

3. Lockstep constant-time simulations

We present a simple method for proving preservation of observational non-interference, corresponding to lockstep simulations. Moreover, we instantiate our method to constant folding and spilling. Refinements of the method to manysteps and general simulations are provided in the next sections.

3.1. Non-cancellation of leakage

Our main results are based on the assumption the relation ψ is non-cancelling w.r.t. leakage. Informally, non-cancellation states that the leakage of an execution uniquely determines the leakage of all its individual steps. Formalizing this property requires some care. Technically, we define the set $\mathcal{L}_{\mathrm{atomic}} = \{l \in \mathcal{L} \mid \exists a, a' \in \mathcal{S}.\ a \xrightarrow{l} a'\}$ of atomic leakages. A relation $\psi \subseteq \mathcal{L} \times \mathcal{L}$ is then said to be non-cancelling iff:

Program counter:
$\mathcal{L} \triangleq \mathsf{list}(\mathbf{B} + \{\bullet\})$, $\cdot \triangleq ++$, $\epsilon \triangleq \mathsf{nil}$

$$\lambda_{\mathsf{skip}} = \lambda_{\mathsf{loop}} = \lambda_{\mathsf{a}}(a, z) = \lambda_{\mathsf{v}}(x, z) = [\bullet] \quad \lambda_{\mathsf{if}}(z) = [z \neq 0]$$

Constant-time:
$\mathcal{L} \triangleq \mathsf{list}(\mathbf{B} + (\mathcal{A} \times \mathbf{Z}) + \{\bullet\})$, $\cdot \triangleq ++$, $\epsilon \triangleq \mathsf{nil}$

$$\begin{array}{ll} \lambda_{\mathsf{skip}} = \lambda_{\mathsf{loop}} = [\bullet] & \lambda_{\mathsf{if}}(z) = [z \neq 0] \\ \lambda_{\mathsf{a}}(a, z) = [(a, z)] & \lambda_{\mathsf{v}}(x, z) = [x] \end{array}$$

Figure 4: Non-cancelling program counter and constant-time

- for every $t_1, t'_1 \in \mathcal{L}_{\text{atomic}}$ and $t_2, t'_2 \in \mathcal{L}$,

$$t_1 \cdot t'_1 = t_2 \cdot t'_2 \implies t_1 = t_2 \wedge t'_1 = t'_2$$

- for every $t \in \mathcal{L}_{\text{atomic}}$ and $t' \in \mathcal{L}$, $t \cdot t' \neq \epsilon$

The non-cancellation property entails the following useful property.

Lemma 4. *Assume ψ is non-cancelling. Then for every $t_1, \dots, t_m, t'_1, \dots, t'_n \in \mathcal{L}_{\text{atomic}}$,*

$$t_1 \cdot \ldots \cdot t_m = t'_1 \cdot \ldots \cdot t'_n \Rightarrow m = n \wedge \bigwedge_{1 \leq i \leq n} t_i = t'_i$$

It turns out that none of leakage models from Figure 3 satisfies the non-cancelling condition. However, one can easily define alternative leakage models that verify the non-cancelling condition, and which yield equivalent notions of non-interference. For instance, Figure 4 presents alternative models for the program counter and constant-time policies; informally, the non-cancelling leakage models are obtained by replacing ϵ leakages by a leakage $[\bullet]$, where $\bullet$ is a distinguished element, to record that one execution step has been performed.

3.2. General framework

We start by recalling the standard notion of simulation from compiler verification. In the simplest (lockstep) setting, one requires that the simulation relation relates one step of execution of S with one step execution of C, as shown in Figure 5a, in which black represents the hypotheses and red the conclusions. The horizontal arrows represent one step execution of S from state a to state b, and one step execution of C from state α to state β. The relation $\cdot \approx \cdot$ relates execution states of the source and of the target program.

Definition 5 (Lockstep simulation)**.** $\approx$ is a *lockstep simulation* when:

- for every source step $a \to b$, and every target state α such that $a \approx \alpha$, there exist a target state β and a target execution $\alpha \to \beta$ such that end states are related: $b \approx \beta$;
- for every input parameter i, we have $S(i) \approx C(i)$;
- for every source and target states b and β such that $b \approx \beta$, we have b is a final source state iff β is a final target state.

The following lemma follows from the assumption that all our languages have a deterministic semantics.

Lemma 5. *Assume that $\approx$ is a simulation. Then for every target execution step $\alpha \to \beta$ and safe source state a such that $a \approx \alpha$, there exists a source execution step $a \to b$ such that $b \approx \beta$.*

Our method is based on constant-time simulations, a new proof technique adapted from the simulation technique in compiler verification. Whereas simulations are proved by 2-dimensional diagram chasing, constant-time simulations are proved by 3-dimensional diagram chasing. Figure 5b illustrates the definition of constant-time simulation, for the lockstep case. It introduces relations $\cdot \equiv_S \cdot$ and $\cdot \equiv_C \cdot$ between source and target states, depicted with a triple line in the diagram. Horizontal arrows represent one step executions (as before), but we now consider two executions at source level and two executions at target level.

Definition 6 (Lockstep 2-simulation)**.** $(\equiv_S,\equiv_C)$ is a lockstep 2-simulation with respect to $\approx$ iff

- For every source steps $a \xrightarrow{t} b$ and $a' \xrightarrow{t} b'$ such that $a \equiv_S a'$ and for every target steps from $\alpha \xrightarrow{\tau} \beta$ and $\alpha' \xrightarrow{\tau'} \beta'$ such that $a \approx \alpha$ and $a' \approx \alpha'$ and $\alpha \equiv_C \alpha'$ and $b \approx \beta$ and $b' \approx \beta'$, we have $b \equiv_S b'$ and $\beta \equiv_C \beta'$ and $\tau = \tau'$;
- For every pair of input parameters $i,\ i'$ s.t. $\phi\ i\ i'$, we have $S(i) \equiv_S S(i')$ and $C(i) \equiv_C C(i')$.

The notion of constant-time simulation is tailored to make the $\cdot \equiv \cdot$ relation inductive, and to yield preservation of (both variants of) observational non-interference. This is captured by the next theorem, where ϕ is the relation on input parameters —it is useful to think about ϕ as low-equivalence between memories—and $\equiv_S$ and $\equiv_C$ are two relations on source and target states–it is useful to think about $\equiv_S$ and $\equiv_C$ as equivalence of code pointer, i.e. the two states point to the same instruction.

Theorem 1 (Preservation of constant-time policy)**.** *Assume that leakage is non-cancelling. Let S be a safe source program and C be the target program obtained by compilation. If S is TIONI (resp. TSONI) w.r.t. $(\phi, =)$ then C is TIONI (resp. TSONI) w.r.t. $(\phi, =)$, provided the following holds:*

1) *$\approx$ is a lockstep simulation;*
2) *$(\equiv_S, \equiv_C)$ is lockstep 2-simulation w.r.t. $\approx$.*

Proof sketch. Consider two target executions

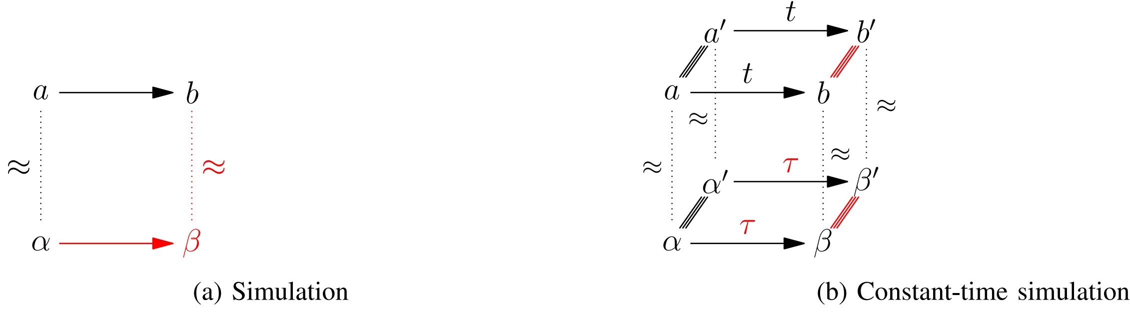
$$\alpha_1 \xrightarrow{\tau_1} \ldots\ldots \xrightarrow{\tau_m} \alpha_{m+1} \qquad \alpha'_1 \xrightarrow{\tau'_1} \ldots\ldots \xrightarrow{\tau'_n} \alpha'_{n+1}$$

starting in related states, i.e. $\phi\ \alpha_1\ \alpha'_1$, and such that α_{m+1} and α'_{n+1} are final states. We must show that $m = n$ and $\tau_i = \tau'_i$ for every $1 \leq i \leq n$. By safety of S and 1) and Lemma 5, there exist source executions

$$a_1 \xrightarrow{t_1} \ldots\ldots \xrightarrow{t_m} a_{m+1} \qquad a'_1 \xrightarrow{t'_1} \ldots\ldots \xrightarrow{t'_n} a'_{n+1}$$

such that $\phi\ a_1\ a'_1$ and $a_i \approx \alpha_i$ for every $1 \leq i \leq m+1$ and $a'_i \approx \alpha'_i$ for every $1 \leq i \leq n+1$. Moreover, by 1) a_{m+1} and a'_{n+1} are final states, and by 2) $\alpha_1 \equiv_C \alpha'_1$ and $a_1 \equiv_S a'_1$. By the constant-time property of S, and the non-cancelling property of leakage, $m = n$ and $t_i = t'_i$ for every $1 \leq i \leq n$. We next reason by induction on i, applying 2) to conclude that $\alpha_i \equiv_C \alpha'_i$ and $a_i \equiv_S a'_i$ and $\tau_i = \tau'_i$, as desired. □

Theorem 1 reduces proving constant-time of preservation to proving the existence of a simulation $\approx$ and a constant-time simulation $(\equiv_S, \equiv_C)$ relative to $\approx$. Furthermore, both goals can be proved



(a) Simulation

(b) Constant-time simulation

Figure 5: Lockstep simulations

independently. This separation of concerns limits the proof effort and support modular proofs. In particular, when the compiler is already proved correct (by showing a simulation $\approx$), one only needs to show that there exists a constant-time simulation w.r.t. $\approx$.

3.3. Discussion

Theorem 1 and its subsequent generalizations (Theorem 4 and Theorem 6) are broadly applicable to settings where both the source and target policies are strengthenings of the program counter policies. This limitation is best explained by an analogy with unwinding lemmas, a standard technique for proving information flow policies. However, we stress that the analogy is given for explanatory purposes only; in particular, our method for proving preservation of observational non-interference is modular and oblivious to the choice of a relation ϕ and of the method used for proving observational non-interference.

Informally, unwinding lemmas come in two flavours: "locally preserves" unwinding lemmas show that two executions started in related states yield related states, where the notion of related states is given by a low-equivalence relation akin to ϕ, and "step-consistent" unwinding lemmas show that under some conditions, one step execution yields related states, i.e. $a \to a'$ implies that a and a' are low-equivalent. "Step-consistent" unwinding lemmas are used for reasoning about diverging control flow, i.e. when programs branch on secrets, and are not required when executions have the same control flow, as imposed by the program counter policy. Our methods only consider the equivalent of "locally preserves" unwinding lemmas, and are therefore restricted to policies that entail program counter security. It is certainly possible to extend our method to provide a counterpart to the "step-preserving" policies. However, it remains an open question whether the general framework could still be instantiated to a broad class of program optimizations.

3.4. Examples

3.4.1. Constant folding. We illustrate the general scheme for proving preservation of the constant-time policy, taking *constant folding* as example. This simple optimization searches for constant expressions and replaces them by their value in the program text. Technically, constant folding traverses the program and replaces expressions op e_1 e_2 by simpler expressions if e_1 or e_2 evaluate to distinguished constants. The simplification rules for multiplication work as follows: if both subexpressions compile to known values n_1 and n_2 respectively, the compilation is the constant $n_1 \times n_2$, if one of the arguments compiles to the constant 1 then the result is the compilation of the other. The most interesting case is if one of the argument compiles to 0. In this case, the result is the constant 0 (independently of the other argument).

We use the lockstep simulation technique to prove that constant-folding preserves the constant-time policy. Our first step is to prove that the constant-folding satisfies the lockstep diagram for simulation. To this end, we consider the relation $a \approx \alpha$ is defined by

$$[\![a.\mathsf{cmd}]\!] = \alpha.\mathsf{cmd} \wedge a.\mathsf{env} = \alpha.\mathsf{env}$$

Lemma 6. *The relation $\approx$ is lockstep simulation invariant.*

The proof of this lemma is based on the fact that if an expression e has a semantics in a given environment ρ (i.e. $[e]_\rho = n$) then its compilation has the same semantics, $[[\![e]\!]]_\rho = n$.

Our next step is to prove that the transformation satisfies the lockstep diagram for constant-time simulation. To this end, we use for the $\equiv$ relations the equality of commands $a \overset{\mathsf{c}}{\equiv} a'$ defined by $a.\mathsf{cmd} = a'.\mathsf{cmd}$.

Lemma 7. $(\overset{\mathsf{c}}{\equiv}, \overset{\mathsf{c}}{\equiv})$ *is a lockstep 2-simulation w.r.t. $\approx$.*

The proof of this lemma is based on the fact that if an expression e has a (leaky) semantics in both environments and the leakages coincide (i.e. $[e]_\rho = n$ and $[e]_{\rho'} = n'$ and $\mathrm{leak}(e, \rho) = \mathrm{leak}(e, \rho')$), then the compilation of e generates the same leakage in both environments, i.e.

$$\mathrm{leak}([\![e]\!], \rho) = \mathrm{leak}([\![e]\!], \rho')$$

It follows that constant folding preserves the constant-time property.

Theorem 2. *Constant-folding preserves the constant-time policy.*

3.4.2. Register allocation. Register allocation is a compilation pass that maps an unbounded set of variables into a finite set of registers. In order to preserve the semantics of programs, it is often necessary that the target program stores the value of some variables on the stack, effectively turning variable accesses into memory accesses. Formally, register allocation produces for every program point a mapping from program variables to registers or stack variables (interpreted as an integer denoting the relative position in the stack). Finding optimal assignments that minimize register spilling is a hard problem, and is commonly solved using translation validation. Specifically, computing the assignment is performed by an external program, and a verified checker verifies that the assignment is compatible with the semantics of programs. Formally, we modify the language and its semantics to distinguish between regular variables r and stack variables k . The notion of state changes accordingly, i.e. the environment is now a partial map from registers and stack position to values. The semantics of the language is modified accordingly; in particular, each read/write into the stack leaks its position.

$$\begin{aligned}
&\mathrm{leak}(k, \rho) = [k] \\
&\mathrm{leak}\{k = e; c, \rho\} = \mathrm{leak}(e, \rho) \ \ {+}{+} \ [k]
\end{aligned}$$

We let σ be the assignment output by register allocation.

Proving the correctness of register allocation is relatively easy. The proof relies the correctness of the liveness analysis which underlies register allocation. Preservation of the constant time policy is more interesting, because spilling introduces new memory reads and writes. The crucial observation is that the addresses leaked by spilling do not depend on the memory, since they are constant offset relative to the top of the stack. Thus, the proof of the 2-simulation diagram does not pose any specific difficult. For both proofs, we use $a \approx \alpha$ defined by

$$[\![a.\mathsf{cmd}]\!] = \alpha.\mathsf{cmd} \wedge \forall x.\ a.\mathsf{env}(x) = \alpha.\mathsf{env}(\sigma(x))$$

Figure 6: Manysteps simulations

Theorem 3. *$\approx$ is lockstep simulation invariant. $(\overset{c}{\equiv}, \overset{c}{\equiv})$ is a lockstep 2-simulation relative to $\approx$. Variable spilling preserves the constant-time policy.*

Remark 1. Following [5], our formalization separates register allocation in two steps. The first step performs some form of variable renaming and defines for every program point a mapping from source variables to target variables. The second step performs spilling, and defines a single, global, mapping from variables to variables or stack variables. Furthermore, the mapping must be the identity for variables that are mapped to variables.

This proof can be extended to a more complex language with function calls and stack pointer. The difficulty here is that a stack address will not be a constant k but relative to the top of the stack $\mathsf{stk} + k$. Then, when proving preservation of constant-time we have to show that introduced leaks are equal in two different executions, i.e. $\mathsf{stk}_1 + k = \mathsf{stk}_2 + k$ where stk_i correspond to the value of the stack pointer in execution i. So we have to ensure equality of stack pointers. This can be done by a small modification of $\equiv$ predicate of the target language, so that it imposes equality of commands and of stack pointers. Since the value of the stack pointer only depends on the control flow of the program and that preservation of constant-time already requires equality of the control flow, establishing equality of stack pointers adds no difficulty.

4. Manysteps constant-time simulations

The requirement of lockstep execution is often too strong in practice. For illustrative purposes, consider the nonsensical transformation that replaces every atomic instruction i by $\mathsf{skip}; i$. Every single execution step of the source program will correspond to two execution steps of the target program (so it does not correspond to the lockstep simulation). There exist alternative notions of simulations which relax the requirement on lockstep executions by allowing more than one step of execution (i.e. manysteps), as shown on Figure 6a. We show that an equivalent relaxation exists for 2-simulations.

4.1. Framework

Recall that the constant-time simulation diagram considers two instances of the simulation diagram. It is not enough to simply consider two pairs of traces satisfying the previous diagram. To understand how this is an issue, assume a compilation pass (similar to our add skip transformation) that transforms the fictitious program "ℓ: GOTO ℓ" into "ℓ: NOP; GOTO ℓ". Every two steps of target execution returns

to the same state. A simulation relation will necessarily relate the source state to some target state α, and given the hypotheses of the constant-time simulation diagram ($a \approx \alpha$, $a \to a$, and $\alpha \to^+ \alpha$) it is generally not possible to tell how many loop iterations separate the two occurrences of the target state α (hence to prove that any two such target executions have the same length).

To overcome this issue, the simulation diagram is refined to predict how many steps of the target will correspond to each step of the source. This information is usually implicit in the simulation proof; we only make it explicit to be used in the constant-time simulation diagram.

Formally, we introduce a function $\mathsf{num\text{-}steps}(a, \alpha)$ which, assuming that a and α are two reachable states related by the simulation relation, predicts how many steps of the target semantics are to be run starting from α to close the manysteps simulation diagram.

Definition 7 (Manysteps simulation). $\approx$ is a manysteps simulation w.r.t. $\mathsf{num\text{-}steps}$ when:

- for every source steps $a \to b$, and every target state α such that $a \approx \alpha$, there exist a target state β and target execution $\alpha \to^n \beta$, where $n = \mathsf{num\text{-}steps}(a, \alpha)$, such that end states are related: $b \approx \beta$;
- for every input parameter i, we have $S(i) \approx C(i)$;
- for every source and target states b and β such that $b \approx \beta$, we have b is a final source state iff β is a final target state.

Given such a simulation diagram, it is possible to build the constant-time simulation diagram that universally quantifies over two instances of this diagram, as depicted on Figure 6b. The diagram reads as follows.

Definition 8 (Manysteps 2-diagram). A pair of relations ($\equiv_S$, $\equiv_C$) is a *manysteps 2-simulation w.r.t.* $\approx$ *and* $\mathsf{num\text{-}steps}$ iff the following holds: for all $a,\ a',\ b,\ b',\ \alpha,\ \alpha',\ \beta,\ \beta',\ t,\ \tau,\ \tau'$ such that

- initial states are related $a \equiv_S a'$ and $\alpha \equiv_C \alpha'$;
- $a \xrightarrow{t} b$ and $a' \xrightarrow{t} b'$;
- $\alpha \xrightarrow{\tau}{}^{n} \beta$ and $\alpha' \xrightarrow{\tau'}{}^{n'} \beta'$ where $n = \mathsf{num\text{-}steps}(a, \alpha)$ and $n' = \mathsf{num\text{-}steps}(a', \alpha')$;
- the simulation relation holds $a \approx \alpha$, $a' \approx \alpha'$, $b \approx \beta$, and $b' \approx \beta'$

we have

- equality of leakage $\tau = \tau'$ and $n = n'$,
- final states are related $b \equiv_S b'$ and $\beta \equiv_C \beta'$.

This definition, together with a condition on initial states, enable us to define the manysteps constant-time simulation.

Definition 9 (Manysteps 2-simulation). A pair of relations ($\equiv_S$, $\equiv_C$) is a *manysteps 2-simulation relative to* $\approx$ *w.r.t.* $\mathsf{num\text{-}steps}$ when:

1) $(\equiv_S, \equiv_C)$ satisfy the manysteps 2-diagram w.r.t. $\approx$ and $\mathsf{num\text{-}steps}$;
2) for every pair of input parameters $i,\ i'$ s.t. $\phi\ i\ i'$, we have $S(i) \equiv_S S(i')$ and $C(i) \equiv_C C(i')$.

Theorem 4 (Constant-time preservation from manysteps 2-simulation). *Assume that leakage is non-cancelling. Let S be a safe source program and C be the target program obtained by compilation. If S is TIONI (resp. TSONI) w.r.t. $(\phi, =)$ then C is TIONI (resp. TSONI) w.r.t. $(\phi, =)$, provided the following holds, for a given num-steps function that is strictly positive:*

1) *$\approx$ is a manysteps-simulation w.r.t. num-steps;*
2) *($\equiv_S$, $\equiv_C$) is a manysteps 2-simulation relative to $\approx$ w.r.t. num-steps.*

4.2. Example: Expression flattening

The flattening of expressions models the transformation to a 3-address code format: each expression is split in a sequence of assignments and a final expression such that at most one operator appears in each expression. This actually fixes the evaluation order of the sub-expressions. Since the evaluation of an expression is done, after transformation, in several steps, the leakage corresponding to an expression is spread among the leakages of these steps.

The transformed program may use additional variables to store the values of some sub-expressions. Therefore, the set of program variables is extended with names for such temporary values.

The procedure that flattens an expression e is written $F(e)$: it returns a pair (p, e') where p is a *prefix* command and e' an expression such that the evaluation of p followed by the evaluation of e' yields the same value as the evaluation of e; moreover, the evaluation of p only modifies fresh temporary variables. The transformation $[\![\cdot]\!]$ of a program applies F to each expression e occurring in that program and inserts the corresponding prefix just before, as follows—we note $(p, e') = F(e)$:

$$\begin{array}{lcl} [\![x = e]\!] & = & p; x = e' \\ [\![\mathsf{if}\ e\ c_1\ c_2]\!] & = & p; \mathsf{if}\ e'\ [\![c_1]\!]\ [\![c_2]\!] \\ [\![\mathsf{loop}\ c_1\ e\ c_2]\!] & = & \mathsf{loop}\ ([\![c_1]\!]; p)\ e'\ [\![c_2]\!] \end{array}$$

Note here that the loop structure of our language conveniently allows not to duplicate the sequence p, as arbitrary instructions can be executed before the loop guard.

The correctness of this transformation states that the source and compiled programs agree on the values of the non-temporary variables. To this end, the relation $a \approx \alpha$ between states is defined as follows:

$$[\![a.\mathsf{cmd}]\!] = \alpha.\mathsf{cmd} \land \forall x,\ a.\mathsf{env}(x) = \alpha.\mathsf{env}(x)$$

where x denotes a original program variable (i.e. not a temporary variable introduced by the compilation). The target command is the result of the compilation of the source command and the source and target memories agree on the values of the non-temporary variables.

To show that this relation is a simulation, we must predict the number of target steps corresponding to each source step. For each states a and α, we define $\mathsf{num\text{-}steps}(a, \alpha)$ as $1 + n$, where n is the length of the prefix of the expression that is evaluated during the step starting in a (if any). By construction, this function is strictly positive. To prove that this pass preserves constant-time, we build a manysteps 2-simulation diagram.

Theorem 5. *The relation $\approx$ is a manysteps simulation w.r.t* $\mathsf{num\text{-}steps}$*. ($\overset{c}{\equiv}$, $\overset{c}{\equiv}$) is a manysteps 2-simulation relative to $\approx$ w.r.t* $\mathsf{num\text{-}steps}$*. Expression flattening preserves the constant-time policy.*

The main argument is that if the evaluations of two expressions produce the same leakage, then evaluations after flattening also produces the same leakage. Notice that this transformation may choose any evaluation strategy for the expressions, and that needs not be related to the order that appears in the definition of the leakage of expressions.

5. General constant-time simulations

5.1. Framework

In this section, we relax the condition of 2-simulations by allowing the number num-steps to be zero or positive. In addition, we strengthen the assumption of 2-simulation so that one can use that

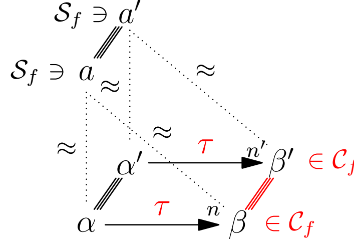


Figure 7: Final 2-diagram

the full source program (and not only the current step) is constant-time. Formally, the definition of the manysteps 2-diagram is complemented with additional hypotheses: initial source states a and a' are reachable in S and are a constant-time pair of states; initial target states α and α' are reachable in C. Finally, we relax the condition on final state in the simulation: when the source execution terminates the target execution is allowed to take a few more steps.

To allow num-steps to be zero, the simulation diagram features an additional constraint: there should be no infinite sequence of source steps that are simulated by an empty sequence of target steps (every source state in the sequence being in relation with a single one target state). This constraint is usually formalized by means of a *measure* of source states which strictly decreases whenever the corresponding target state stutters.

Definition 10 (General simulation)**.** Given a relation $\approx$ between source an target states, a function $|\cdot|$ from source states to natural numbers[2], and a function num-steps from pairs of source and target states to natural numbers; we say that $\approx$ is a *general simulation w.r.t.* $|\cdot|$ *and num-steps* when:

- for every source step $a \to b$ and every target state α such that $a \approx \alpha$, there exist a target state β and a target execution $\alpha \to^n \beta$ where $n = \text{num-steps}(a, \alpha)$ such that end states are related: $b \approx \beta$;
- for every source step $a \to b$ and every target state α such that $a \approx \alpha$ and $\text{num-steps}(a, \alpha) = 0$, the measure of the source state strictly decreases: $|a| > |b|$;
- for every final source state a and every target state α such that $a \approx \alpha$, there exist a *final* target state β and a target execution $\alpha \to^n \beta$ where $n = \text{num-steps}(a, \alpha)$ such that end states are related: $a \approx \beta$.

To allow the target execution to terminate after the source execution, we introduce the following variant of the 2-simulation, depicted on Figure 7.

Definition 11 (Final 2-diagram)**.** A pair of relations ($\equiv_S$, $\equiv_C$) satisfy the *final constant-time diagram w.r.t.* $\approx$ *and num-steps* when the following holds: for all a, a', α, α', β, β', τ, τ' such that:

- initial states are related $a \equiv_S a'$, $\alpha \equiv_C \alpha'$;
- initial source states are final $a, a' \in \mathcal{S}_f$;
- there are two target executions $\alpha \xrightarrow{\tau}{}^n \beta$ and $\alpha' \xrightarrow{\tau'}{}^{n'} \beta'$ where $n = \text{num-steps}(a, \alpha)$ and $n' = \text{num-steps}(a', \alpha')$;
- the simulation relation holds $a \approx \alpha$, $a' \approx \alpha'$, $a \approx \beta$, and $a' \approx \beta'$

we have

- equality of leakage $\tau = \tau'$ and $n = n'$;

2. Any ordered set satisfying the ascending chain condition would be suitable.

$$\begin{aligned}
[\![\textsf{if }0\ c_1\ c_2]\!] &= [\![c_2]\!] \\
[\![\textsf{if }b\ c_1\ c_2]\!] &= \textsf{if }b\ [\![c_1]\!]\ [\![c_2]\!] && \text{if } b \neq 0 \\
[\![\textsf{loop }c_1\ 0\ c_2]\!] &= [\![c_1]\!] \\
[\![\textsf{loop }c_1\ b\ c_2]\!] &= \textsf{loop }[\![c_1]\!]\ b\ [\![c_2]\!] && \text{if } b \neq 0
\end{aligned}$$

Figure 8: Removal of trivial (false) branches

- end states are related and final $\beta \equiv_C \beta',\ \beta, \beta' \in \mathcal{C}_f$.

Definition 12 (General 2-simulation)**.** Given a relation $\approx$, a function $|\cdot|$, and a function num-steps as above, we say that the pair of relations $(\equiv_S, \equiv_C)$ is a *(general) 2-simulation* when:

1) $(\equiv_S, \equiv_C)$ satisfy the manysteps 2-diagram w.r.t. $\approx$ and num-steps;
2) for every pair of input parameters i, i' s.t. $\phi\ i\ i'$, we have $S(i) \equiv_S S(i')$ and $C(i) \equiv_C C(i')$;
3) for all related source states $a \equiv_S a'$, none or both of them are final: $a \in \mathcal{S}_f \iff a' \in \mathcal{S}_f$;
4) $(\equiv_S, \equiv_C)$ satisfy the final 2-diagram w.r.t. $\approx$ and num-steps.

Theorem 6 (Constant-time preservation from general 2-simulation)**.** *Assume that leakage is non-cancelling. Let S be a safe source program and C be the target program obtained by compilation. If S is TIONI (resp. TSONI) w.r.t. $(\phi, =)$ then C is TIONI (resp. TSONI) w.r.t. $(\phi, =)$, provided the following holds, for given num-steps and $|\cdot|$ functions:*

1) *$\approx$ is a general simulation w.r.t. $|\cdot|$ and num-steps;*
2) *$(\equiv_S, \equiv_C)$ is a general 2-simulation w.r.t. $\approx$, $|\cdot|$, and num-steps.*

Remark 2. In the Coq formalization, we use slightly more convenient definitions in which more hypotheses are available: all considered states are reachable, initial source states (a and a') are a constant-time pair of states. This enables to do more modular proofs.

5.2. Examples

5.2.1. Dead branch elimination. In this section, we present a transformation that we designed for the purpose of illustration: a more general and less artificial version will be discussed in the following section. This transformation removes conditional branches whose conditions are trivially false. More precisely, the compilation function $[\![\cdot]\!]$ is defined as depicted on Figure 8. Assignments are kept unchanged; the if instructions whose conditions are trivially false (i.e., the literal 0 is used as guard) are replaced by their else branch (recursively compiled); similarly, loop instructions whose guard is false are removed: only the first part of the loop body is kept (recursively compiled).

To justify the correctness of this transformation, we define a relation $a \approx \alpha$ between source state a and target state α as: $\alpha = \{[\![a.\textsf{cmd}]\!], a.\textsf{env}\}$; the target command is the compilation of the source command, and both states have the same environment.

Interestingly, this transformation may *remove* execution steps: the ones corresponding to the evaluation of the trivially false conditions and the unfolding of the removed loops. Therefore, the $\cdot \approx \cdot$ relation does not satisfy the lockstep simulation diagram; it satisfies however the more general simulation diagram. To that end, we define the num-steps function as 1 for all states excepted if the current instruction is a

conditional on false (if or loop), we also define a measure $|a|$ of source execution states by taking the full size of $a.\mathsf{cmd}$, such that this measure strictly decreases when the target takes no step.

Theorem 7. *$\approx$ is general simulation invariant. $(\overset{c}{\equiv}, \overset{c}{\equiv})$ is a general 2-simulation relative to $\approx$, num-steps and $|\cdot|$. Dead branch elimination preserves the constant-time policy.*

5.2.2. Constant propagation. A slightly more interesting variant of the previous transformation is constant propagation: a static analysis prior to the compilation pass infers at each program point, which variables hold a statically known constant value; then using this information, each expression can be simplified (as in the constant folding transformation described in § 3.4.1) and the branches that are trivial after this simplification can be removed.

This transformation, as many other common compilation passes, relies on the availability of a *flow-sensitive* analysis result: some information must be attached to every program point. As usual, since our language has no explicit program points, we enrich its syntax with annotations. More precisely, each instruction gets one annotation, except the loop which gets two: one that is valid at the beginning of each loop iteration, and one that is valid when evaluating the loop guard. The small-step semantics, when executing a loop, introduces an if and a loop. The annotations to put on the next iteration may depend on the purpose of these annotations; therefore, the semantics is parametrized by a annot-step function which describes how to compute the annotations of a loop at the next iteration, yielding the following rule for the execution of the loop instruction (decorated letters k figure the annotations):

$$\{\mathsf{loop}^{k_1}_{k_2}\ c_1\ b\ c_2, \rho\} \to \{c_1; \mathsf{if}^{k_2}\ b\ (c_2; \mathsf{loop}^{k'_1}_{k'_2}\ c_1\ b\ c_2)\ \mathsf{skip}, \rho\}$$

where $(k'_1, k'_2) = \mathsf{annot\text{-}step}(k_1, k_2)$.

In this particular case of constant propagation, we assume that the source program is annotated with partial mappings from variables to constant values (integers). Those annotations are generated by a first pass of analysis and are certified independently. Given this information, the compilation may simplify expressions, remove if whose guard is trivial, and remove loops whose guard is false.

However, the compilation only transforms constructs that syntactically appear in the program source code and does not operates on the constructs that are in the execution state as a result of the semantic execution of the program. In particular, the compilation will not transform a trivial if that results from the execution of a loop whose guard is trivially true. More generally, a compilation pass may apply different transformations to similar pieces of code depending on some heuristic, and these heuristics should be irrelevant to the correctness argument. To overcome this issue we reuse the annotation facility to be able to statically determine how each source instruction is transformed. The compiler generates two programs: an annotated version of the source[3] and its compiled version. The compiler adds a boolean flag to the source to tell whether a branch is eliminated or not. Therefore, trivial branches that only appear during the execution and are not visible to the compiler will not have this flag set and the num-steps function can predict that the corresponding execution step is preserved. The flag part of k_2 is false indicating that the if is preserved; the annot-step function is the identity. The num-steps function is defined following the same idea of the previous example. As well, to prove that execution does not stutter for ever we use the size of the command. The $\approx$ predicate ensures equality of environments and that the target code come from the compilation of the source taking into account the flags.

Theorem 8. *$\approx$ is general simulation invariant. $(\overset{c}{\equiv}, \overset{c}{\equiv})$ is a general 2-simulation relative to $\approx$, num-steps and $|\cdot|$. Constant-propagation preserves the constant-time policy.*

3. which is proved equivalent to the original program up to annotation

$$\frac{c_1 \sim c_1' \quad c_2 \sim c_2'}{\mathsf{loop}^0\ c_1\ b\ c_2 \sim \mathsf{loop}\ c_1'\ b\ c_2'}$$

$$\frac{c_1 \sim c_1' \quad c_2 \sim c_2' \quad \mathsf{loop}^n\ c_1\ b\ c_2 \sim c'}{\mathsf{loop}^{n+1}\ c_1\ b\ c_2 \sim c_1'; \mathsf{if}\ b\ (c_2'; c')\ \mathsf{skip}}$$

Figure 9: Specification of loop peeling

5.2.3. Loop peeling. Loop peeling is an optimization that unrolls the first iterations of loops. It is a good example where annot-step is not the identity function but counts the number of loop iterations.

How many iterations are actually peeled may depend on heuristics which should not be visible in the correctness proof (nor in the simulation argument). In this case, instead of proving one particular compilation scheme, we define a relation between source and target commands that captures various possible ways to perform the transformation. This relation is written $\sim$ and defined on Figure 9. Each *source* loop instruction is annotated with a number stating how many iterations of this loop are peeled. For these annotations to remain consistent during the execution, this counter is decremented on each iteration: formally, $\mathsf{annot\text{-}step}(n, a) = (\max(0, n-1), a)$.

For this transformation, the $\approx$ predicate is equality of environments and $\sim$ of commands. The num-steps function is equal to 1, except on loop with a non-zero annotation for which it is 0. The $|\cdot|$ function is the number of nested loop in the first instruction.

Theorem 9. *$\approx$ is simulation invariant. $(\overset{c}{\equiv}, \overset{c}{\equiv})$ is a 2-simulation relative to $\approx$, num-steps and $|\cdot|$. Loop peeling preserves the constant-time policy.*

5.2.4. Linearization. Linearization transforms programs into lists of instructions with explicit labels. The linear language features (only) two instructions: assignment "$x = e$" of the value of an expression e to a l-value x; and conditional branching "jnz b n", which continues execution at (relative) position n when expression b evaluates to a non-zero value or falls through the next instruction otherwise. When the condition is syntactically 1, we simply write "goto n". In this language, a program is a list of instructions, and execution starts at position zero, i.e., at the beginning of said list. The execution state is a triple made of the program counter, the current environment, and the whole program.

In the language, the leakage of a step correspond to the leakage of the expressions and the value of the next program point. This is analogous to leaking the boolean value of conditional jumps.

The transformation $[\![\cdot]\!]$ from the structured language from previous sections to linear is described in Figure 10. It introduces forward jumps at the end of else branches (to bypass the corresponding then branches) and at the beginning of loops (to bypass the second body before the first iteration). These jumps do not correspond to any particular instruction in the source. To define the $\approx$ relation for the correctness proof, we always allow the target execution to perform (any number of) such jumps. The technical difficulty of the proof comes from the fact that source instruction will be in relation with many target instructions, some of them correspond to the compilation of the original instruction but some of them come from its context; breaking the locality of the proof.

Moreover, there may be a final sequence of jumps at the end of the target execution. This implies that the target execution may be delayed a little bit after the source execution finished. This is why we use the additional diagram for final states in the 2-simulation. Furthermore, since in the linear language the full program is part of the running state and never change, it is convenient to show this invariant once and for all. This is where the reachability hypotheses are useful.

$$\llbracket x = e \rrbracket = x = e$$
$$\llbracket \mathsf{if}\ b\ c_1\ c_2 \rrbracket = \mathsf{jnz}\ b\ n_2; \llbracket c_2 \rrbracket; \mathsf{goto}\ n_1; \llbracket c_1 \rrbracket$$
$$\text{// where } n_2 = |\llbracket c_2 \rrbracket| + 2 \text{ and } n_1 = |\llbracket c_1 \rrbracket| + 1$$
$$\llbracket \mathsf{loop}\ c_1\ b\ c_2 \rrbracket = \mathsf{goto}\ n_2; \llbracket c_2 \rrbracket; \llbracket c_1 \rrbracket; \mathsf{jnz}\ b\ n$$
$$\text{// where } n_2 = |\llbracket c_2 \rrbracket| + 1 \text{ and } n = -|\llbracket c_2 \rrbracket; \llbracket c_1 \rrbracket|$$

Figure 10: Linearization

To establish the 2-simulation, we introduce a relation $\stackrel{L}{\equiv}$ between states that claims that the program point is the same. The proof of the 2-diagram follows without surprises. The technicalities due to the change in representation (from a structured language to an unstructured one) are dealt with using the same lemmas as for the correctness proof.

The num-steps function is defined as the number of forward jump from the current target program counter plus 1 if the current source instruction is not a loop. The $|\cdot|$ function is the number of nested loops in the first source instruction.

Theorem 10. *The relation $\approx$ is a simulation w.r.t num-steps. ($\equiv$, $\stackrel{L}{\equiv}$) is a 2-simulation relative to $\approx$ w.r.t. num-steps and $|\cdot|$. Linearization preserves the constant-time policy.*

5.2.5. Common branches factorization. In these last two sections, we consider a new kind of transformation where the order of evaluation of expressions/instructions is changed. The first transformation consists in factorizing the common prefix of conditional branches. In other terms, the transformation will replace a statement of the form $\mathsf{if}\ e\ (h; c_1)\ (h; c_2)$ by $h; \mathsf{if}\ e\ c_1\ c_2$, when the evaluation of e is not affected by the evaluation of c. This allows to reduce the size of the code. To prove the correctness, we need to establish the general simulation for a given relation $\approx$. When the if instruction on e is executed in the source program, the h instructions should be first executed in the target program before executing the conditional on e. Similarly, when the c instructions are executed in the source program, they have been already executed in the target program, so nothing should be done. For the correctness of the compiler the difficulty is that environment of the source program and of the target program are desynchronised at some point. For 2-simulation the difficulty is that leaks of h will appear in the target trace before they appear in the source. In particular the trace t corresponding to the leak of the if e does not contain the leaks of h, so we should be able to prove that the leaks of h will be equal in both evaluations because the source program in constant-time.

The notion of simulation used for this transformation $a \approx \alpha$ is defined by:

$$\exists h,\ a.\mathsf{cmd} \stackrel{h}{\sim} \alpha.\mathsf{cmd} \wedge \{a.\mathsf{env}, h\} \rightarrow^* \{\alpha.\mathsf{env}, \mathsf{skip}\}$$

The predicate $c \stackrel{h}{\sim} c'$ is presented Figure 11. Morally c is the code of the source program, c' of the target program, and h is the sequence of instructions which have been already executed in the target program but are still to be executed in the source. The notion of simulation requires that the evaluation of h in the source environment terminates on the target environment. When $h = \mathsf{skip}$ we omit it.

If h is skip, the rules (omitted in the figure) say that the predicate $\sim$ is structural. To simplify the proof, we assume that source program is annotated with numbers. For assignments and loops, the

$$\frac{c_1 \overset{h}{\sim} c_1' \qquad c_2 \overset{h}{\sim} c_2' \qquad \mathsf{terminating}(h) \qquad \mathsf{write}(h) \cap \mathsf{read}(e) = \emptyset}{\mathsf{if}^{|h|+1}\ e\ c_1\ c_2 \sim h; \mathsf{if}\ e\ c_1'\ c_2'} \qquad \frac{c \overset{h}{\sim} c'}{i^0; c \overset{i;h}{\sim} c'}$$

Figure 11: $\approx$ predicate for common branches elimination

number indicates the value of the num-steps function needed by the simulation. For conditional, if the number is not 0, it corresponds to the number of instructions which have been factorized out by the compiler. The code of each branch should be related by $\overset{h}{\sim}$, h should be terminating[4] and should not modify the variables read by e (this ensures that the evaluation of e can swap with the evaluation of h).

If h is not skip, see the last rule, it means that the target program has already executed h and the source program should catch up with the target. This means that the source evaluation step corresponds to no execution step in the target program; this is why the annotation in the source is 0.

The num-steps function cannot be fully inferred statically (i.e., at compile time), because h may contain conditionals and so the number of steps to execute depends on the execution environment and of which branches are taken. It is not a problem with our proof technique since the num-steps function is parametrized by the source and the target states. The termination condition ensures that the evaluation of h will terminate in a finite number of steps[5]. The $|\cdot|$ function is the size of the source instruction.

Theorem 11. *$\approx$ is general simulation with w.r.t. $|\cdot|$ and num-steps . $(\overset{c}{\equiv}, \overset{c}{\equiv})$ is a general 2-simulation relative to $\approx$. Common branches factorization preserves the constant-time policy.*

5.2.6. Switching instructions. Switching instructions of a program is a very common optimization. It can be used together with common branches to improve its efficiency, but its most common use is for instruction scheduling. It is a typical example of transformation that depends on heuristics (which can be different for each target architecture) which should not be visible in the correctness proof. To that end, we simply prove a checker taking a source program and a target program and returns true if they are equal up to a *valid* permutation inside basic blocks (list of assignments), but the control-flow is still the same. The notion of *valid* permutation should ensure that the semantic of the program is unchanged. For example, $c_1; c_2$ can be transformed into $c_2; c_1$ if variables read and written by c_2 are not written by c_1 (and vice versa), furthermore both instructions should terminate[6]. To be able to define the $\approx$ relation we encounter the same difficulty as for common branches factorization: the source and target states are not synchronized. Figure 12 defines the $\sim$ predicate used to define $\approx$. As for common branches factorization it is parametrized by a code h representing instructions already executed in the target and that remain to be executed in the source. Intuitively, $c \overset{h}{\sim} c'$ should be interpreted has c is a *valid permutation* of $h; c'$. Some rules (not shown) allow to perform transformation under context, the two presented rules define the permutation. The predicate swap ensures that the two instructions can be safely swapped without changing the semantic of the program. The num-steps function return directly the annotation of the current instruction, the $|\cdot|$ is the size of the instruction.

4. Our formalization requires that h is a sequence of assignments and conditionals. It would be possible to support loops for which the compiler is able to prove termination.

5. Without that condition it is not clear that the correctness of the compiler can be expressed/proved using a small step simulation diagram.

6. Termination of c_1 is needed to ensure preservation of non-interference: if c_1 is non-terminating then $c_1; c_2$ can be a TSONI program whereas $c_2; c_1$ may not.

$$\frac{c \overset{h_1;h_2}{\sim} c' \qquad \mathsf{swap}(i, h_1)}{i^0; c \overset{h_1;i;h_2}{\sim} c'} \qquad \frac{c \overset{h_1;h_2}{\sim} c' \qquad \mathsf{swap}(i, h_1; h_2)}{i^{|h_2|}; c \overset{h_1}{\sim} h_2; i; c'}$$

Figure 12: $\approx$ predicate for switching instructions

Theorem 12. *$\approx$ is a general simulation w.r.t. $\mathsf{num\text{-}steps}$ and $|\cdot|$. $(\overset{c}{\equiv}, \overset{c}{\equiv})$ is a 2-simulation relative to $\approx$. Switching instructions preserves the constant-time policy.*

6. Related work

Many recent works address the challenge of verifying constant-time implementations formally, using established methods such as type systems, abstract interpretation and deductive verification. These methods are applied to source programs [10], assembly programs [6], or intermediate representations (e.g. LLVM IR) [4], [25]. These works do not address the problem of policy-preserving compilation—see however the informal discussion in [4]. Preservation of the constant-time policy by compilation is discussed explicitly in [5], where Almeida *et al* provide an informal argument for the Jasmin compiler; our method provides the means to make their proof formal. Independently, preservation of the constant-time policy by compilation is studied in [24], where Protzenko *et al.* claim that their translation from λow* to C* preserves constant-time. We have been unable to reconstruct their argument, as it uses the usual notion of simulations. However, we believe that the notion of constant-time simulation can also be applied in their setting. Barthe *et al* [7] develop a general method for result-preserving compilation, and use their method to improve the precision of a constant-time analysis at intermediate level. However, their work focuses on preserving the results of alias analyses and does not study preservation of the constant-time policy. Almeida *et al* [3] give a simple translation validation technique to guarantee preservation of security in the program counter model for the CompCert compiler.

Other works have considered preservation of specific information flow policies by compilation. Barthe *et al* [8] define an information flow type system for a concurrent programing language and prove that typable programs are compiled into non-interfering assembly programs, under reasonable assumptions on the scheduler. Laud [17] and Fournet and Rezk [12] define information flow type systems for a core imperative language, and prove that programs are compiled into cryptographically secure implementations. Murray *et al* [21] prove preservation of value-dependent noninterference for a concurrent imperative language. Their proof uses coupling invariants, which are related to 2-simulations. However, they do not explicitly address the problem of preserving 2-properties by standard compilation passes.

Motivated by developing foundations for secure compilation, Patrignani and Garg [22] explore preservation of hyperproperties as an alternative to full abstraction. However, they do not propose general proof techniques for proving that specific compilation passes preserve a given hyperproperty.

For lack of space, we ignore other broad areas of research, including quantitative analysis of side-channels, and new computation paradigms, such as ORAM or hardware-protected mechanisms, that can potentially protect against side-channel attacks.

7. Conclusion

We have developed a general method for proving preservation of software-based side-channel countermeasures by compilation, and provided instantiations to representative compiler optimizations. In our experience, proving preservation of constant-time policy is sometimes simpler than proving correctness of the transformation.

Additionally, we have formalized both the general methods and their instantiations using the Coq proof assistant. The overall development is over 8,000 lines of Coq code. The proofs of optimizations range from 250-300 lines (constant propagation and spilling) to 750-800 lines (code motion and expression flattening).

A natural target for future work is to apply our methods to existing verified compilers. Although our work is carried for a rather simple language, we are confident that our selection of optimizations covers the main difficulties that can appear when proving preservation of the constant-time policy for C like languages. In particular, we are confident that our methods are applicable to the Jasmin compiler.

References

[1] Martin R. Albrecht and Kenneth G. Paterson. Lucky microseconds: A timing attack on amazon's s2n implementation of TLS. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 622–643. Springer, 2016.

[2] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 526–540. IEEE Computer Society, 2013.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1217–1230. ACM, 2013.

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 53–70. USENIX Association, 2016.

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 24th ACM Conference on Computer and Communications Security* (*CCS*), 2017.

[6] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1267–1279. ACM, 2014.

[7] Gilles Barthe, Sandrine Blazy, Vincent Laporte, David Pichardie, and Alix Trieu. Verified translation validation of static analyses. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 405–419. IEEE Computer Society, 2017.

[8] Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of multithreaded programs by compilation. In Joachim Biskup and Javier Lopez, editors, *Computer Security - ESORICS 2007, 12th European Symposium On Research In Computer Security, Dresden, Germany, September 24-26, 2007, Proceedings*, volume 4734 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2007.

[9] Daniel J. Bernstein. Cache-timing attacks on aes, 2005. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[10] Sandrine Blazy, David Pichardie, and Alix Trieu. Verifying constant-time implementations by abstract interpretation. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*, volume 10492 of *Lecture Notes in Computer Science*, pages 260–277. Springer, 2017.

[11] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*. USENIX Association, 2003.

[12] Cédric Fournet and Tamara Rezk. Cryptographically sound implementations for typed information-flow security. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 323–335. ACM, 2008.

[13] Daniel Genkin, Luke Valenta, and Yuval Yarom. May the fourth be with you: A microarchitectural side channel attack on several real-world applications of curve25519. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 845–858. ACM, 2017.

[14] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 490–505. IEEE Computer Society, 2011.

[15] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. When constant-time source yields variable-time binary: Exploiting curve25519-donna built with MSVC 2015. In *15th International Conference on Cryptology and Network Security* (*CANS*), pages 573–582, 2016.

[16] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[17] Peeter Laud. Semantics and program analysis of computationally secure information flow. In David Sands, editor, *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2028 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2001.

[18] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 549–564. USENIX Association, 2016.

[19] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *Security and Privacy*. IEEE Computer Society, 2015.

[20] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In Dongho Won and Seungjoo Kim, editors, *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers*, volume 3935 of *Lecture Notes in Computer Science*, pages 156–168. Springer, 2005.

[21] Toby C. Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah. Compositional verification and refinement of concurrent value-dependent noninterference. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 417–431. IEEE Computer Society, 2016.

[22] Marco Patrignani and Deepak Garg. Secure compilation and hyperproperty preservation. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 392–404. IEEE Computer Society, 2017.

[23] Colin Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[24] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified Low-Level Programming Embedded in F*. In *ICFP*, 2017.

[25] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 110–120. ACM, 2016.

[26] Arjun Suresh, Erven Rohou, and André Seznec. Compile-time function memoization. In *26th International Conference on Compiler Construction*.

[27] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptology*, 23(1):37–71, 2010.

[28] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: A timing attack on openssl constant time RSA. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 346–367. Springer, 2016.

Appendix

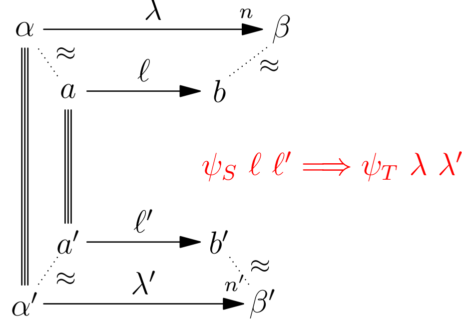


Figure 13: Lockstep 2-simulation

Beyond equality of leakage

Our methods can be adapted to an arbitrary relation ψ on leakage, provided it satisfies two assumptions. The first assumption is that $\cdot$ preserves ψ, i.e.

$$\psi\ t_1\ t_1' \wedge \psi\ t_2\ t_2' \implies \psi\ (t_1 \cdot t_2)\ (t_1' \cdot t_2')$$

The second assumption is a generalization of the non-cancellation property. Formally, we require the following two properties:

- for every $t_1, t_1' \in \mathcal{L}_{\text{atomic}}$ and $t_2, t_2' \in \mathcal{L}$,
$$\psi\ (t_1 \cdot t_1')\ (t_2 \cdot t_2') \implies \psi\ t_1\ t_2 \wedge \psi\ t_1'\ t_2'$$
- for every $t, t' \in \mathcal{L}_{\text{atomic}}$ and $t'' \in \mathcal{L}$,
$$\neg(\psi\ (t \cdot t')\ t'')$$

These two iterms guarantee the following property: for every $t_1, \ldots, t_m, t_1', \ldots, t_n' \in \mathcal{L}_{\text{atomic}}$,

$$\psi\ (t_1 \cdot \ldots \cdot t_m)\ (t_1' \cdot \ldots \cdot t_n') \Rightarrow m = n \wedge \bigwedge_{1 \le i \le n} \psi\ t_i\ t_i'$$

In order to extend Theorem 1 and its generalizations to arbitrary relations ψ, one must also adjust the definition of 2-simulations to consider two relations ψ_S on source leakage and ψ_T on target leakage. For instance, in the case of lockstep 2-simulation, as illustrated on Figure 13, one must require that for every source steps $a \xrightarrow{t} b$ and $a' \xrightarrow{t'} b'$ such that $a \equiv_S a'$ and $\psi_S\ t\ t'$ for every target steps from $\alpha \xrightarrow{\tau} \beta$ and $\alpha' \xrightarrow{\tau'} \beta'$ such that $a \approx \alpha$ and $a' \approx \alpha'$ and $\alpha \equiv_C \alpha'$, we have $b \equiv_S b'$ and $\beta \equiv_C \beta'$ and $\psi_T\ \tau\ \tau'$.

Note that this generalization can be useful when considering a combination of the bounded-relative cost and the program counter policy, specifically, a policy that enforces in addition to the program counter policy that the difference in execution cost between any two corresonding execution steps is upper bounded by some integer k. A complete treatment of this example, and more generally a study of preservation of observational non-interference for ressource-based policies is left for future work.