Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time"

Gilles Barthe, Benjamin Grégoire, Vincent Laporte

To cite this version:

Gilles Barthe, Benjamin Grégoire, Vincent Laporte. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". CSF 2018 - 31st IEEE Computer Security Foundations Symposium, Jul 2018, Oxford, United Kingdom. hal-01959560

HAL Id: hal-01959560

https://hal.archives-ouvertes.fr/hal-01959560

Submitted on 18 Dec 2018

Secure compilation of side-channel countermeasures: the case of cryptographic “constant-time”

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 for cryptographic “constant-time”, a popular countermeasure against cache-based timing attacks. We present a general method, based on the notion of constant-time-simulation, for proving that a compilation pass preserves the constant-time countermeasure. Using the Coq proof assistant, we verify the correctness of our method and of several representative instantiations.

I. 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 [20], [12]: 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], [28], [31], [17], [23], [32], [16]. Similarly, data timing channels exploit the timing variability of specific operations—i.e. operations whose execution time depends on their arguments [24].

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 [25] and the constant-time[1] [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 expert 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.

[1] The terminology “constant-time” is well established in the cryptographic community, but may be confusing for a broader audience, so we shall often (but not always) use the expression “cryptographic constant-time” to minimize risks of confusion.

To address this problem, we propose a general method for proving that cryptographic constant-time is preserved by compilation. Our method is based on constant-time-simulation (CT-simulation), which adapts to our problem the usual notion of simulation from compiler verification. As for simulations, CT-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 cryptographic constant-time for a representative set of compiler optimizations. Our proofs are formally verified using the Coq proof assistant.[2]

Overall, our work lays previously missing theoretical foundations for preservation of cryptographic constant-time, a popular software-based side-channel countermeasures by compilation.

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

- we provide a general method to prove preservation of cryptographic constant-time by compilation;
- we study common classes of compilation passes and prove that they preserve cryptographic constant-time;
- we provide mechanized proofs of correctness of our method, and of the instantiations to specific optimizations.

II. Cryptographic constant-time

Timing attacks are common and very efficient methods to break cryptographic schemes. The most famous one is certainly the attack on the square-and-multiply algorithm used in modular exponentiation. The algorithm that should compute $x^k \mod p$ can be implemented as follows (in pseudo-code):

```
r = 1;
for(i = base - 1; 0 <= i; --i) {
    r = (r * r) mod p;
    if ((k >> i) & 1) r = (r * x) mod p;
}
```

At each iteration r contains the value of $x^{k/2^i}$, each loop iteration squares r and if the bit at i is 1 then r is multiplied by x. If an attacker can measure the time taken by each iteration of the loop, it can distinguish between the iteration where the tested bit of k is 0 or 1. A solution to fix the problem is to systematically execute the second multiplication and then correct the value of r using a conditional move.

```
for(i = base - 1; 0 <= i; --i) {
    r = (r * r) mod p;
    r' = (r * x) mod p;
    r = ((k >> i) & 1) ? r' : r;
}
```

The last instruction can be implemented using a linear combination if the architecture does not provide a constant-time cmov instruction. Importantly, the modified implementations are secure in the program counter model, i.e. their control flow does not depend on secrets.

However, program counter security does not always suffice to protect implementations, as monitoring of shared resources can be exploited by a malicious party to recover information. For instance, cache attacks exploit the latency between cache misses and cache hits to observe whether memory accesses have been performed. Early examples of cache attacks were demonstrated by Percival [28] on an OpenSSL implementation of RSA, and by Bernstein [9] and Osvik, Shamir and Tromer [31] on implementations of AES based on lookup tables.

One popular solution is then to require that memory accesses, as well as control-flow, should not depend of secret data. This policy, known as "constant-time" policy in the cryptography literature, has become a de facto standard for cryptographic implementations—as well as the terminology, which is somewhat misleading, as no explicit reasoning about execution time is involved in the formal definition of constant-time program. Recently, Barthe et al [5] show that constant-time implementations are protected against cache attacks in an idealized model of virtualization.

This is not to say that the notion of cryptographic constant-time is an absolute guarantee against side-channel attacks. For instance, elementary operations such as multiplication and division are not constant-time on many popular architectures. Worse, recent attacks exploit speculative execution to retrieve confidential information from constant-time implementations [19]. However, the constant-time policy remains a useful guideline for writing secure implementations.

In this paper, we study the impact of compiler optimizations on the constant-time policy. It is well-known that compilers may turn source programs that satisfy cryptographic constant-time into target programs that are not constant-time. For a concrete example, consider the code snippet:

```
int mask = -b;
x = (y & mask) | (x & ~mask);
```

This snippet, which has the same effect as move instruction ($x = b?y : x$), is trivially constant-time. Unfortunately the clang compiler version 5.0 using flags -O2 -m32 -march=i686 compiles it into assembly code equivalent to if (b) x = y.

More generally, a classical example of program where the constant-time policy is broken by compilation is string equality. String equality is generally implemented by a library function and it is generally viewed as a non-leaking function at source level; however, it can be compiled to a loop which early exits when two bytes differ. Thus, the low-level program may reveal the first position where the two input bitstrings differ, and hence is not constant-time—assuming that the bitstrings are secret.

Lazy operations are another good example where branching instructions may be introduced during compilation. For example, in x = f(k) && g(z) might be compiled into x = f(k); if x then

[2] The whole development is available at https://sites.google.com/view/ctpreservation

x = g(z); thus a branching on a secret may appear and the final program may not be constant-time.

For a more recent example, Kaufmann et al. [18] 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.

III. Problem statement

A. Observational non-interference

We focus on programs that carry an explicit notion of leakage. In our setting, leakage is modelled as lists of atomic leakages. We let $\mathcal{L}$ denote the list of atomic leakages, and use $\cdot$ to denote concatenation of lists. We also use $[a]$ to denote the list with a single element a, and ϵ to denote the empty list.

We model the behavior of programs using an instrumented operational semantics 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 shall sometimes write $a \to a'$ when t is irrelevant.

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

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

The assumption of deterministic semantics simplifies the formal treatment, and is fully compatible with the intended application domain—in particular, all recent languages for writing high-speed cryptographic software have a deterministic semantics.

We define multi-step execution $a \xrightarrow{t}{}^{+} a'$ by the clauses:

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

We also define n-step execution similarly:

$$\frac{}{a \xrightarrow{\epsilon}{}^{0} a} \qquad\qquad \frac{a \xrightarrow{t} a' \qquad a' \xrightarrow{t'}{}^{n} a''}{a \xrightarrow{t \cdot t'}{}^{n+1} a''}.$$

Observational non-inteference is defined for complete executions. Therefore, we must introduce notions of initial and final states.

Final states are modelled by a distinguished subset $\mathcal{S}_f$ of final states, such that $a \xrightarrow{t} a'$ 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' and leakage t such that $a \xrightarrow{t} a'$. We write $a \Downarrow_t$ iff there exists a final state $a' \in \mathcal{S}_f$ such that $a \xrightarrow{t}{}^{+} a'$.

In view to instantiate our general framework to a standard imperative language, where all the initial states of program P are of the form $\{\rho, P\}$, where ρ is an environment, we assume given a type $\mathcal{I}$ of input parameters and we see a program P as a function mapping input parameters to initial states. Therefore the set of initial states of program P is defined as $P(\mathcal{I})$. It is important to note that the set $\mathcal{I}$ of inputs shall be shared by all languages involved in the compilation chain considered in this paper.

We next define the notion of observationally non-interfering program.

Definition 1 (Observationally non-interfering program)**.** A program P is observationally non-interfering w.r.t. a binary relation ϕ on states, written $P \models \mathsf{ONI}(\phi)$, iff for all states $a, a' \in P(\mathcal{I})$ and $b, b' \in \mathcal{S}$ and $t, t' \in \mathcal{L}$ and $n \in \mathbf{N}$,

$$a \xrightarrow{t}{}^{n} b \wedge a' \xrightarrow{t'}{}^{n} b' \wedge \phi\ a\ a' \implies$$
$$t = t' \wedge (b \in \mathcal{S}_f \iff b' \in \mathcal{S}_f).$$

Our notion of observational non-inteference entails a weaker but more intuitive termination-insensitive notion. Specifically, if P is observationally non-interfering w.r.t. a relation ϕ, then for all states $a, a' \in P(\mathcal{I})$ and $t, t' \in \mathcal{L}$,

$$a \Downarrow_t \wedge a' \Downarrow_{t'} \wedge \phi\ a\ a' \implies t = t'.$$

B. Secure compilation

For convenience, we restrict our attention to safe programs.

Definition 2 (Safety)**.** 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 $a \to a'$. We say that a program P is safe iff for every $a \in P(\mathcal{I})$, for every $a' \in \mathcal{S}$ such that $a \xrightarrow{t}{}^{+} a'$, a' is safe.

The problem addressed in this paper is an instance of secure compilation.

Definition 3 (Security-preserving compiler)**.** Assume given source and target languages, and let ϕ and ϕ' be binary relations on source and target states. Let $[\![\cdot]\!]$ be a compiler from source to target programs. $[\![\cdot]\!]$ is security-preserving for (ϕ, ϕ') iff for every program P:

$$P \models \mathsf{ONI}(\phi) \wedge \mathsf{safe}(P) \overset{?}{\implies} [\![P]\!] \models \mathsf{ONI}(\phi')$$

C. Discussion

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. Such a strategy can be implemented using an adaptation of the standard notion of simulation, which we describe in the next section.

However, this strategy fails for most optimizations—and moreover, source and target languages do not even need to support the same notion of leakage. The failure of the naive strategy forces us to consider an alternative strategy inspired from unwinding lemmas, a standard technique for proving that programs satisfy an information flow policies.

Informally, unwinding lemmas are parametrized by an unwinding relation and come in two flavours: *locally preserves* unwinding lemmas show that one-step executions started in two states related by the unwinding relation yield states that are related by the unwinding relation, and *step-consistent* unwinding lemmas show that under some conditions, one-step execution yields a state related to the original state, i.e. $a \to a'$ implies that a and a' are related by the unwinding relation. Step-consistent unwinding lemmas are used for reasoning about

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

where x ranges over variables, a ranges over arrays and e ranges over expressions.

Fig. 1: Minimal language

diverging control flow, i.e. when programs branch on secrets, and are not required when executions have the same control flow. Our method considers only the case of *locally preserves* unwinding lemmas, and provide sufficient conditions for the unwinding relation and equality of leakage to be preserved by compilation. This suffices for the observational non-interference policies studied in this article.

It is certainly possible to extend our method to provide a counterpart to step-preserving unwinding lemmas. However, it remains an open question whether the general framework could still be instantiated to a broad class of program optimizations.

IV. Setting

In this section we instantiate observational non-interference to a specific language and leakage model.

A. Programming language

Usually, compilers use many intermediate languages during the compilation. Some transformations go from one language to another, while others stay within the same language. While our methodology applies to transformations with different input and output languages, for the sake of simplicity, we try to share as much as possible the input and output language. The only transformation where the language changes is the linearization where the input language is a structured language while the output is a list of basic instructions with jumps.

For most of the transformations, we consider a minimal imperative language shown in Figure 1. The type of input parameters is the set of environments. The language features a loop 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.

B. 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 $\lambda_{\mathsf{a}} : \mathcal{A} \times \mathbf{Z} \to \mathcal{L}$.

C. 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 components of a state. We define initial and final states to be respectively of the form $\{P, \rho\}$ and $\{\mathsf{skip}, \rho\}$, where ρ ranges over environments.

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 conditional, the leakage depends of the branch taken.

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

Lemma 1. *For all states a, b, b', if $a \xrightarrow{t} b$ and $a \xrightarrow{t'} b'$, then $b = b'$ and $t = t'$.*

D. Leakage models

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; Figure 3 summarizes the main leakage models.

- step-counting policy: the number of execution steps is leaked—modelled as the length of a list over the unit type (whose element is noted •);
- program counter policy: control flow is leaked. Leakage is a list of boolean values containing all guards evaluated during this execution;
- memory obliviousness: memory accesses are leaked. Leakage is a list of memory addresses accessed during this execution (but not their value);
- 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. In all examples in this paper, we

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

$$
\begin{array}{rcl}
\{x = e, \rho\} & \xrightarrow{\mathsf{leak}(e,\rho)\cdot\lambda_{\mathsf{v}}(x,[e]_\rho)} & \{\mathsf{skip}, \rho\{x \leftarrow [e]_\rho\}\} \\
\{a[e_1] = e_2, \rho\} & \xrightarrow{\mathsf{leak}(e_1,\rho)\cdot\mathsf{leak}(e_2,\rho)\cdot\lambda_{\mathsf{a}}(a,[e_1]_\rho)} & \{\mathsf{skip}, \rho\{a[[e_1]_\rho] \leftarrow [e_2]_\rho\}\} \\
\{\mathsf{if}\ e\ c_1\ c_2, \rho\} & \xrightarrow{\mathsf{leak}(e,\rho)\cdot\lambda_{\mathsf{if}}([e]_\rho)} & \{c_1, \rho\} \quad \text{if } [e]_\rho \neq 0 \\
\{\mathsf{if}\ e\ c_1\ c_2, \rho\} & \xrightarrow{\mathsf{leak}(e,\rho)\cdot\lambda_{\mathsf{if}}([e]_\rho)} & \{c_2, \rho\} \quad \text{if } [e]_\rho = 0 \\
\{\mathsf{skip}; c_2, \rho\} & \xrightarrow{\lambda_{\mathsf{skip}}} & \{c_2, \rho\} \\
\{\mathsf{loop}\ c_1\ e\ c_2, \rho\} & \xrightarrow{\lambda_{\mathsf{loop}}} & \{c_1; \mathsf{if}\ e\ (c_2; \mathsf{loop}\ c_1\ e\ c_2)\ \mathsf{skip}, \rho\}
\end{array}
\qquad
\frac{\{c_1, \rho\} \xrightarrow{t} \{c_1', \rho'\}}{\{c_1; c_2, \rho\} \xrightarrow{t} \{c_1'; c_2, \rho'\}}
$$

Fig. 2: Instrumented semantics of the while language

use the non-cancelling variant (see below) of this leakage model;

- cost obliviousness: execution cost is leaked. Leakage is a list of numbers, representing the cost of each instruction;
- 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;
- 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.

In all these cases, we require that leakage is equal in two traces that start from related states. One can weaken these policies in multiple ways, for instance by requiring equality of the overall execution cost (obtained by summing the execution cost of each individual instruction), or by requiring that the difference of leakage at each individual step (or the global leakage) does not exceed a given upper bound.

Definition 4. A program P is cryptographic constant-time w.r.t. ϕ iff it is observationally non-interfering w.r.t. ϕ in the constant-time leakage model.

E. Non-cancellation of leakage

Our main results are based on the assumption that leakage is non-cancelling. Informally, non-cancellation states that the leakage of an execution uniquely determines the leakage of all its individual steps, and that the equality of the leakages of two executions entails the pairwise equality of the leakages of each of their steps.

It turns out that some leakage models from Figure 3 do not satisfy the non-cancelling condition. However, in some cases, one can easily define alternative leakage models that verify the non-cancelling condition, and which yield equivalent notions of non-interference. For instance, the program counter and constant-time policies can be made non-cancelling by replacing ϵ leakages by a leakage $[\bullet]$, where $\bullet$ is a distinguished element, to record that one execution step has been performed.

Step-counting:
Atomic leakages: $\{\bullet\}$

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

Program counter:
Atomic leakages: $\mathbf{B}$

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

Memory obliviousness:
Atomic leakages: $\mathcal{X} + (\mathcal{A} \times \mathbf{N})$

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

Constant-time:
Atomic leakages: $\mathbf{B} + (\mathcal{A} \times \mathbf{Z})$

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

Size non-interference:
$\mathcal{L} \triangleq \mathsf{size}$

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

Cost obliviousness:
Atomic leakages: $\mathbf{N}$

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

In the first two models only commands leak. In the last two models only expressions leak.

Fig. 3: Leakage models

F. Relations on initial states

The relations ϕ considered in the literature generally represent low equivalence, and are 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 $i \in \mathbf{N}$.

V. Lockstep constant-time simulations

We present a simple method for proving preservation of cryptographic constant-time, 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.

A. 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 a source program S with one step execution of its compiled version C, as shown in Figure 4a, 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 step $\alpha \to \beta$ such that end states are related: $b \approx \beta$;
- for every input parameter i, we have $S(i) \approx C(i)$;
- for all 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 2. *Assume that $\approx$ is a simulation. Then for all 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 4b 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 CT-simulation)**.** ($\equiv_S$,$\equiv_C$) is a lockstep CT-simulation with respect to $\approx$ iff

- For all source steps $a \xrightarrow{t} b$ and $a' \xrightarrow{t} b'$ such that $a \equiv_S a'$ and for every pair of target steps $\alpha \xrightarrow{\tau} \beta$ 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 stable by reduction, and to yield preservation of the constant-time policy. 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)**.** *Let S be a safe source program and C be the target program obtained by compilation. If S is constant-time w.r.t. ϕ then C is constant-time w.r.t. ϕ, provided the following holds:*

1) *$\approx$ is a lockstep simulation;*
2) *$(\equiv_S, \equiv_C)$ is lockstep CT-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 2, 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, $m = n$ 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 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. $\approx$.

B. Examples

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

$$\begin{tikzcd} a \arrow[r] \arrow[d, dash, dotted, "\approx"'] & b \arrow[d, dash, dotted, "\approx"] \\ \alpha \arrow[r] & \beta \end{tikzcd}$$

(a) Simulation

$$\begin{tikzcd} & a' \arrow[rr, "t"] \arrow[dl, equal] \arrow[dd, dash, dotted, "\approx"' near start] & & b' \arrow[dl, equal] \arrow[dd, dash, dotted, "\approx"] \\ a \arrow[rr, "t"] \arrow[dd, dash, dotted, "\approx"'] & & b \arrow[dd, dash, dotted, "\approx" near end] & \\ & \alpha' \arrow[rr, "\tau"] \arrow[dl, equal] & & \beta' \arrow[dl, equal] \\ \alpha \arrow[rr, "\tau"] & & \beta & \end{tikzcd}$$

(b) Constant-time simulation

Fig. 4: Lockstep simulations

respectively, the compilation is the constant $n_1 \times n_2$; otherwise, 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 3. *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 4. $(\overset{\mathsf{c}}{\equiv}, \overset{\mathsf{c}}{\equiv})$ *is a lockstep CT-simulation w.r.t.* $\approx$.

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

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

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

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

2) Register allocation and spilling: 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 use a distinguished array that is not used in the source program to model the stack. We let σ be the assignment output by register allocation: it maps each source variable to either a variable or a position in the stack array.

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 at a constant offset relative to the top of the stack. Thus, the proof of the CT-simulation diagram does not pose any specific difficulties. For both proofs, we use $a \approx \alpha$ defined by

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

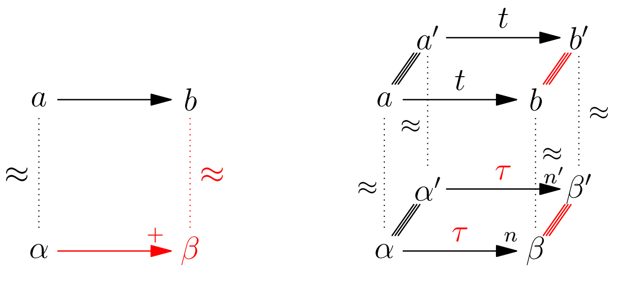
Theorem 3. $\approx$ *is lockstep simulation invariant.* $(\overset{\mathsf{c}}{\equiv}, \overset{\mathsf{c}}{\equiv})$ *is a lockstep CT-simulation relative to* $\approx$*. Variable spilling preserves the constant-time policy.*

Remark 1. Following [4], 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 a 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 the $\equiv$ 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.

VI. 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



(a) Simulation

(b) Constant-time simulation

Fig. 5: Manysteps simulations

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 5a. We show that an equivalent relaxation exists for CT-simulations.

A. 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 ($a \approx \alpha$, $a \rightarrow a$, and $\alpha \rightarrow^{+} \alpha$) 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 $\textsf{num-steps}(a, \alpha)$ 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. $\textsf{num-steps}$ when:

- for all source steps $a \rightarrow b$, and every target state α such that $a \approx \alpha$, there exist a target state β and target execution $\alpha \rightarrow^{n} \beta$, where $n = \textsf{num-steps}(a, \alpha)$, such that end states are related: $b \approx \beta$;
- for every input parameter i, we have $S(i) \approx C(i)$;
- for all 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 5b. The diagram reads as follows.

Definition 8 (Manysteps CT-diagram)**.** A pair of relations ($\equiv_S$, $\equiv_C$) is a *manysteps CT-simulation w.r.t.* $\approx$ *and* $\textsf{num-steps}$ iff the following holds: for all a, a', b, b', α, α', β, β', t, τ, τ' 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 $n = \textsf{num-steps}(a, \alpha)$ and $n' = \textsf{num-steps}(a', \alpha')$;
- 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 CT-simulation)**.** A pair of relations ($\equiv_S$, $\equiv_C$) is a *manysteps CT-simulation relative to* $\approx$ *w.r.t.* $\textsf{num-steps}$ when:

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

Theorem 4 (Constant-time preservation from manysteps CT-simulation)**.** *Let S be a safe source program and C be the target program obtained by compilation. If S is constant-time w.r.t. ϕ then C is constant-time w.r.t. ϕ, provided the following holds, for a given* $\textsf{num-steps}$ *function that is strictly positive:*

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

B. Example: Expression flattening

The flattening of expressions models the transformation to a 3-address code format: each expression is split into 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 CT-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 CT-simulation relative to $\approx$ w.r.t. $\mathsf{num\text{-}steps}$. Expression flattening preserves the constant-time policy.*

The main argument is that if two evaluations of an expression in two environments produce the same leakage, then the evaluations after flattening also produce 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.

VII. General constant-time simulations

A. Framework

In this section, we relax the condition of CT-simulations by allowing the number num-steps to be zero or positive. In addition, we strengthen the assumption of CT-simulation so that one can use that the full source program (and not only the current step) is constant-time. Formally, the definition of the manysteps CT-diagram is complemented with additional hypotheses: initial source states a and a' are reachable in S and are a constant-time pair of states[3]; initial target states α and α' are reachable in C. Finally, we relax the condition on final states in the simulation: when the source execution terminates the target execution is allowed to take a few more steps.

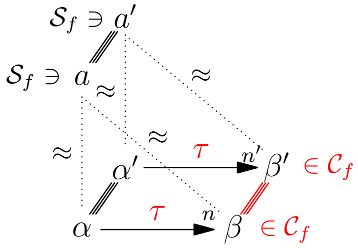


Fig. 6: Final CT-diagram

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 (all source states 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 and target states, a function $|\cdot|$ from source states to natural numbers[4], 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* $\mathsf{num\text{-}steps}$ 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 $n = \mathsf{num\text{-}steps}(a, \alpha)$ 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 $\mathsf{num\text{-}steps}(a, \alpha) = 0$, 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 $n = \mathsf{num\text{-}steps}(a, \alpha)$ 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 CT-simulation, depicted on Figure 6.

Definition 11 (Final CT-diagram)**.** A pair of relations $(\equiv_S, \equiv_C)$ satisfy the *final constant-time diagram w.r.t.* $\approx$ *and* $\mathsf{num\text{-}steps}$ 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 $n = \mathsf{num\text{-}steps}(a, \alpha)$ and $n' = \mathsf{num\text{-}steps}(a', \alpha')$;
- 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 $n = n'$;

[3] Reachable from two states that are related by ϕ through two executions which produce the same leakage.

[4] Any ordered set satisfying the ascending chain condition would be suitable.

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

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

1) $(\equiv_S, \equiv_C)$ satisfy the manysteps CT-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 $C(i) \equiv_C C(i')$;
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 CT-diagram w.r.t. $\approx$ and $\mathsf{num\text{-}steps}$.

Theorem 6 (Constant-time preservation from general CT-simulation)**.** *Let S be a safe source program and C be the target program obtained by compilation. If S is constant-time w.r.t. ϕ then C is constant-time w.r.t. ϕ, provided the following holds, for given $\mathsf{num\text{-}steps}$ and $|\cdot|$ functions:*

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

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 proofs to be more modular.

Remark 3. The final CT-diagram is not strictly necessary. However, it is a very convenient tool to simplify simulation relations. Without this additional diagram, the simulation relation needs special cases to explain the last steps at the end of the target executions, introducing disjunctions and many extra cases in the proofs. This is similar to, e.g., the lock-step diagram: it is subsumed by the general diagram but much more convenient to use, when possible.

B. Examples

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 7. 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: $\alpha = \{[\![a.\mathsf{cmd}]\!], a.\mathsf{env}\}$; the target command is the compilation of the source command, and both states have the same environment.

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

Fig. 7: Removal of trivial (false) branches

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 $\mathsf{num\text{-}steps}$ 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 $a.\mathsf{cmd}$. Therefore, this measure strictly decreases when the target takes no step.

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

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 § V-B1) 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 $\mathsf{annot\text{-}step}$ 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):

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

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 operate 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[5] 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 $\mathsf{num\text{-}steps}$ 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 $\mathsf{annot\text{-}step}$ function is the identity. The $\mathsf{num\text{-}steps}$ 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 CT-simulation relative to $\approx$, $\mathsf{num\text{-}steps}$ and $|\cdot|$. Constant-propagation preserves the constant-time policy.*

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

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

Fig. 8: Specification of loop peeling

3) Loop peeling: Loop peeling is an optimization that unrolls the first iterations of loops. It is a good example where $\mathsf{annot\text{-}step}$ 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 8. 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, $\mathsf{annot\text{-}step}(n, a) = (\max(0, n-1), a)$.

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

Fig. 9: Linearization

For this transformation, the $\approx$ predicate is equality of environments and $\sim$ of commands. The $\mathsf{num\text{-}steps}$ 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 CT-simulation relative to $\approx$, $\mathsf{num\text{-}steps}$ and $|\cdot|$. Loop peeling preserves the constant-time policy.*

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 "$\mathsf{jnz}\ b\ n$", 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 "$\mathsf{goto}\ n$". 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 corresponds 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 $\llbracket \cdot \rrbracket$ from the structured language from previous sections to linear is described in Figure 9. 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 each 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 CT-simulation. Furthermore, since in the linear language

[5] This output source program is proved equivalent to the original program up to annotation.

the full program is part of the running state and never changes, it is convenient to show this invariant once and for all. This is where the reachability hypotheses are useful.

To establish the CT-simulation, we introduce a relation $\overset{L}{\equiv}$ between states that claims that the program point is the same. The proof of the CT-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 $\mathsf{num\text{-}steps}$ 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. $\mathsf{num\text{-}steps}$. ($\overset{c}{\equiv}$, $\overset{L}{\equiv}$) is a CT-simulation relative to $\approx$ w.r.t. $\mathsf{num\text{-}steps}$ and $|\cdot|$. Linearization preserves the constant-time policy.*

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 h. 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 h 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 the environments of the source program and of the target program are desynchronised at some point. For CT-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 $\mathsf{if}\ e$ 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:

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

The predicate $c \overset{h}{\sim} c'$ is presented Figure 10. 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 number indicates the value of the $\mathsf{num\text{-}steps}$ 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[6] and should not modify the variables read by e (this ensures that the evaluation of e can swap with the evaluation of h).

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

Fig. 10: $\approx$ predicate for common branches factorization

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 $\mathsf{num\text{-}steps}$ 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 $\mathsf{num\text{-}steps}$ 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[7]. The $|\cdot|$ function is the size of the source instruction.

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

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[8]. 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

[6] 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.

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

[8] Termination of c_1 is needed to ensure preservation of constant-time.

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

Fig. 11: $\approx$ predicate for switching instructions

synchronized. Figure 11 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.

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

VIII. Towards realistic compilers

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 full-fledged compilers. We discuss the case of the Jasmin and CompCert compilers. The discussion is summarized in Table II and Table I.

A. Jasmin

Jasmin [4] is a framework for building high speed and high assurance cryptographic software, using state-of-the-art methods from software verification and certified compilation. In the intended workflow, applications developed within this framework are written in the Jasmin programming language, which features a carefully chosen combination of high-level features (structured control flow such as loops), low-level abstractions (generic pseudo-assembly) and platform-specific instructions. Programs in the Jasmin language thus retain the familiar flavour of source programs and do not require programmers to develop their applications in less familiar and more restrictive languages—e.g. languages that do not have full support for loops. Jasmin programs are compiled into x64 assembly using the Jasmin compiler. The Jasmin compiler is proved correct relative to big-step semantics of source and assembly programs.

Preservation of the constant-time policy by compilation is discussed explicitly by Almeida *et al.* [4] who provide an informal argument for the Jasmin compiler. We believe that our method provides the means to make their proof formal. Concretely, Figure 12 displays the compilation chain for Jasmin programs. Table II gathers all compilation passes, indicating for each of them how they relate to optimizations studied in this paper. On this account, we are confident that most of the compilation passes from Jasmin could be proved to preserve constant-time. However, the big-step semantics currently used for Jasmin remains a technical hurdle towards this goal.

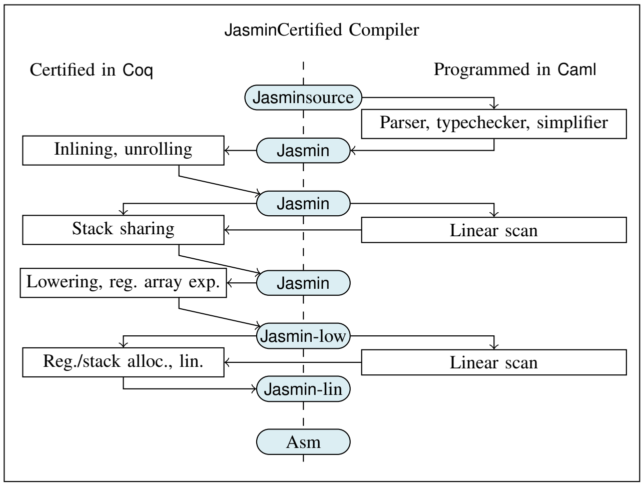


Fig. 12: Jasmin architecture [4].

B. CompCert

CompCert [22] is a verified, moderately optimizing, compiler for C. From a high-level perspective, CompCert compiles C programs to RTL (Register Transfer Language) programs, where programs are represented by their control-flow graphs and are optimized using dataflow analyses, and then to assembly programs, modelled as lists of low-level instructions operating over machine registers and memory locations. However, CompCert actually uses a dozen intermediate languages, each of which is defined by a formal small-step semantics that model normal termination, divergence, abnormal termination and undefined behaviors (such as out-of-bounds array accesses).

CompCert comes with a semantics preservation theorem that is obtained from the correctness of each of the compiler passes. Table I summarizes the passes of the CompCert compiler, indicating for each of them how they relate to optimizations studied in this paper.

Known issues with preservation of the constant-time policy by CompCert include the conversion between integers and floating points, which may introduce conditional branches that are not explicit present in the source program [11], and the compilation of specific 64-bits operations on 32-bits platforms[9].

C. Other languages

We believe that our methodology should also apply to higher-level languages like Java; nonetheless, compilation passes that are specific to these languages – implementation of high-level abstractions, specific optimizations – may require specific arguments, hence further investigation.

[9] Xavier Leroy, personal communication

TABLE I: CompCert compilation passes

name	comment
SimplExpr	Fixes the evaluation order; before this pass, the semantics is non-deterministic; the notion of constant-time is not well defined for the source program.
SimplLocals	This step removes leakage; it is similar to our “register allocation” with trivial cases only.
C#minor gen.	Similar to our “constant folding”.
Cminor gen.	Allocates local variables to the stack; similar to our “spilling” pass.
Selection	Similar to our “constant folding”.
RTL gen.	Combines our “expression flattening” and “linearization” passes.
Tailcall	This pass reorders the behavior of a return instruction before the behavior of a call instruction; it thus has a lot in common with our passes which reorder instructions.
Inlining	These four passes change the leakage in a deterministic fashion, as in our “spilling” pass; they may change the control-flow (call-stack, program counter) and the concrete memory addresses.
Renumber	
Unusedglob	
CleanupLabels	
Constprop, CSE	Similar to our “constant folding” pass.
Deadcode	This pass removes some instructions; showing ct-preservation is a simple instance of the general CT-simulations.
Allocation	Similar to our “spilling” pass.
Tunneling	Part of our “linearization” pass covers this transformation.
Linearize	Similar to our “linearization” pass.
Stacking	Similar to our “spilling” pass.
ASM gen.	the leakage model may change, as in our “linearization” pass.

TABLE II: Jasmin compilation passes

name	comment
Inlining	Changes the control-flow (hence the leakage) in a deterministic fashion.
Unused functions	Preserves leakage.
Loop unrolling	Similar to our “loop peeling”.
Alloc inline assgn	Preserves leakage.
Constant propagation	Combines our “constant folding” and “dead branch elimination” passes.
Dead code elimination	This pass removes some instructions; showing ct-preservation is a simple instance of the general CT-simulations.
Share stack var.	Preserves leakage.
Array init.	This is a special case of “dead code elimination” above.
Reg. array expansion	Preserves leakage.
Lowering	Similar to our “expression flattening”.
Reg. allocation	Preserves leakage (as there is no spilling).
Stack allocation	Similar to our “spilling”.
Linearization	Similar to our “linearization”.
ASM generation	Preserves leakage.

D. Using a compiler preserving the constant-time policy

In order to get a target program that is proved to comply with the constant-time policy using a compiler that is proved to preserve this property, it is required to first prove that the source program is safe (i.e., free of undefined behaviours) and constant-time. Moreover, the precise definition of the constant-time policy, i.e., the leakage model, should be the same for both proofs: the proof that the source program is constant-time; and the proof that the compiler preserves the constant-time property. In this respect, there is a tension between what to expose as libraries (which must be verified as part of the source program) and what to treat as programming language constructs (which must be correctly implemented during compilation). Providing many constant-time primitives makes it easier to write constant-time programs at the source-level but makes harder the tasks of implementing and verifying a compiler which provides constant-time implementation of these primitives.

For instance, should a 64-bit shift operator be a constant-time primitive? In such case, a source-level program could safely use it, even on sensitive inputs; but the implementation of the compiler should carefully enforce that property on all architectures —even on 32-bit architectures without such a primitive readily available— lest it should not be constant-time preserving. Otherwise, this operator could be declared as non-constant-time at the source level, shifting the burden towards the users of the source programming language.

Again, preserving the constant-time policy is not a property of a compiler alone: it also depends on the particular leakage models of the source and target languages and libraries.

IX. Related work

Secure compilation is an active area of research in programming languages and cryptography. For the sake of focus, we concentrate on work that specifically addresses cryptographic constant-time and exclude from our discussion of other broad areas of research, including observational non-interference for other leakage models, quantitative analysis of side-channels, new computation paradigms (that can potentially protect against side-channel attacks), such as ORAM or hardware-protected mechanisms.

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) [3], [30]. These works do not address the problem of policy-preserving compilation—see however the informal discussion in [3]. Independently, preservation of the constant-time policy by compilation is mentioned in [29], in the context of a translation from λow* to C* preserves constant-time. Cauligi *et al.* [13] develop a domain specific language and a compiler that generates constant-time code, and use automated verification tools to check that the generated code is constant-time. 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.

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 programming language and prove that typable programs are compiled into non-interfering assembly programs, under reasonable assumptions on the scheduler. Laud [21] and Fournet and

Rezk [15] define information flow type systems for a core imperative language, and prove that programs are compiled into cryptographically secure implementations. Murray *et al.* [26] prove preservation of value-dependent noninterference for a concurrent imperative language. Their proof uses coupling invariants, which are related to CT-simulations. However, they do not explicitly address the problem of preserving 2-properties by standard compilation passes.

D'Silva and coworkers [14] argue that a compiler correctness proof, done using "source-level" semantics, cannot capture the preservation of security properties. However, we show that there is no need for a "machine-level" semantics for the source language to show that a compiler preserves some security properties. To show that a programming discipline like constant-time yields some security properties on a particular platform indeed requires a precise semantics in which said security properties can be expressed. Nonetheless, to show that compliance with such a discipline is preserved by compilation does not necessarily require, as we have described, to do a proof on the machine-level semantics.

Logical relations and full abstraction are two important tools that are widely used for reasoning about compiler correctness. These are broad notions that could potentially be instantiated or adjusted to reason about preservation of the constant-time property. However, there is no general result about logical relations from which our results would follow. Moreover, there have been concerns about full abstraction as a foundation for secure compilation. In particular, Patrignani and Garg [27] explore preservation of hyperproperties as an alternative to full abstraction. However, the criterion which they use for secure compilation, that they call *trace-preserving compilation*, is overly strong for proving preservation of the constant-time property by optimizing compilers. As we have shown in many examples of this paper, optimizations do not preserve leakage traces (in other words, traces that are preserved by optimizing compilers are not detailed enough to model attackers that monitor side-channels).

X. Conclusion

We have developed a general method for proving preservation by compilation of cryptographic constant-time, a popular software-based side-channel countermeasure against cache-based timing attacks, 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).

Future work includes proving preservation of constant-time policy for more realistic compilers and broadening the scope of our methods to other security notions that can be expressed as observational non-interference policies.

Acknowledgments

This work is partially supported by ONR Grants N000141210914 and N000141512750 and by Google Chrome University.

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, 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.

[4] 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.

[5] 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.

[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] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified compilation of floating-point computations. *J. Autom. Reasoning*, 54(2):135–163, 2015.

[12] 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.

[13] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. FaCT: A flexible, constant-time programming language. In *Secure Development Conference (SecDev)*. IEEE, September 2017.

[14] Vijay D'Silva, Mathias Payer, and Dawn Song. The correctness-security gap in compiler optimization. In *Security and Privacy Workshops* (*SPW*), pages 73–87. IEEE, 2015.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018.

[20] 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.

[21] 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.

[22] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

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

[32] 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.