A Study on the Preservation of Cryptographic Constant-Time Security in the CompCert Compiler

Alix Trieu
Univ Rennes, Inria, CNRS, IRISA
35000, Rennes, France
Email: alix.trieu@irisa.fr

***Abstract*—Cryptographic constant-time programming is an established coding discipline used in cryptography to secure programs against timing attacks. Most, if not all, cryptography library try to adhere to this coding style. The C programming language is oftentimes considered a portable assembly, and is hence used by a great number of cryptography libraries. However, what is executed by the hardware is actual assembly, not C. One can thus wonder whether security properties are preserved through compilation as even formally verified compilers only ensure preservation of observable behaviors.**

We present in this paper how to derive a natural framework to prove preservation of cryptographic constant-time security from simulation based proofs of compiler correctness. We also give insights on how this could be adapted to CompCert.

***Index Terms*—formal verification, Coq proof assistant, constant-time security, timing attacks, CompCert, verified compilation**

I. Introduction

Timing and cache side-channels in critical software are among the most dangerous sources of attacks as they can be exploited remotely. Cryptographic constant-time programming has been lauded as a programming discipline that ensures that these side-channels cannot be exploited. This name is actually a bit of a misnomer, as the program does not run in a constant amount of time, but rather that the variation in execution time is independent of secret information. Indeed, this coding style enforces two rules, branching instructions shall not depend on secret information, and neither shall memory accesses. Imposing that branching instructions do not depend on secrets ensures that whether the different branches have the same execution time, an attacker would still not obtain any information on the secrets. Making sure that memory accesses do not depend on secrets is to ensure that no cache attack can be used.

Though the rules of constant-time security are quite simple to state, it is actually very difficult to get right as shown by the diverse list of attacks that exploit such errors in implementations. Consequently, a number of tools to verify that a code is cryptographically constant-time have been proposed [6, 4, 12]. A large number of these tools [1, 5] target a high-level language such as C as it is the language used in most cryptography libraries. However, no compiler guarantees that the security provided by these tools is preserved through compilation, and some even violates this property. For instance, consider the following C code[1] that implements three different ways to write a "selection" function. `y` is returned if `b` is true, else `x` is returned.

```
unsigned not_constant_time
        (unsigned x, unsigned y, bool b)
{
    if (b) { return y; }
    else { return x; }
}

unsigned constant_time_1
        (unsigned x, unsigned y, bool b)
{ return x + (y - x) * b; }

unsigned constant_time_2
        (unsigned x, unsigned y, bool b)
{ return x ^ ((y ^ x) & (-(unsigned) b)); }
```

The first version is self-explanatory, it returns `y` if `b` is true and `x` otherwise. The second version uses the fact that parameter `b` has type `bool`, which in C, is represented by unsigned integers 0 (false) or 1 (true)[2]. If its value is 1 (true), then the returned value is `x + (y - x)` which is equal to `y`. Otherwise, it returns simply `x` since `(y - x) * 0 = 0`. The third version is more elaborate, it uses bitwise operator XOR `^` and bitwise operator AND `&`. It also exploits the wrap around behavior of unsigned integers and since `b` is either 0 or 1, `-(unsigned) b` becomes either -0 = 0 or the integer which has only 1 as bits ($2^{32}-1$ for 32 bits architectures). The result of the bitwise AND operation `((y ^x) & (-(unsigned) b))` is thus `y ^ x` if `b` is true and 0 otherwise. Finally, since `x ^ (y ^ x) = y` and `x ^ 0 = x`, the function returns the expected result of `y` if `b` is true, and `x` otherwise.

If we consider the boolean parameter to be secret information, the first version is not constant-time as it branches on it, whereas the second and third version are constant-time. However, when compiled for older architectures that do not support conditional

[1]The example is inspired from https://twitter.com/volatile_void/status/957899300322840576

[2]More precisely, it is only true since C99 when `<stdbool.h>` is included

moves such as i386 or i486, the compiler Clang version 7.0.0[3] produces code that is not constant-time. The assembly code generated by the compiler is reproduced below in AT&T syntax.

```
1  not_constant_time: # not constant time
2    movb 12(%esp), %al
3    testb %al, %al
4    jne .LBB0_1
5    leal 4(%esp), %eax
6    movl (%eax), %eax
7    retl
8  .LBB0_1:
9    leal 8(%esp), %eax
10   movl (%eax), %eax
11   retl
12 constant_time_1: # not constant time
13   movb 12(%esp), %al
14   testb %al, %al
15   jne .LBB1_1
16   leal 4(%esp), %eax
17   movl (%eax), %eax
18   retl
19 .LBB1_1:
20   leal 8(%esp), %eax
21   movl (%eax), %eax
22   retl
23 constant_time_2: # not constant time
24   movb 12(%esp), %al
25   movl 4(%esp), %ecx
26   testb %al, %al
27   jne .LBB2_1
28   xorl %eax, %eax
29   xorl %ecx, %eax
30   retl
31 .LBB2_1:
32   movl 8(%esp), %eax
33   xorl %ecx, %eax
34   xorl %ecx, %eax
35   retl
```

We first notice that `not_constant_time` and `constant_time_1` both compile to the exact same code except for the label names as the compiler manages to understand that the multiplication by the boolean `b` is equivalent to testing it. The code works as follows, the value at `esp + 12` represents the third parameter of the function which is the boolean `b` in the source code according to calling conventions and is moved into register `al`. The `testb` instruction then sets the ZF (Zero Flag) flag if `b` is false (i.e. 0) and clears the flag otherwise. If the flag is set, then the `jne` jump at line 4 is taken and the effective address `esp + 8` which represents `y` in the source code is computed and loaded into register `eax` before returning. Otherwise, the flag is cleared, and the jump is not taken, `esp + 4` which represents `x` is similarly computed and loaded into `eax` before returning.

The code is thus not constant-time, as the `jne` jumps at line 4 and 15 depend on whether the previous `testb` instructions set the ZF flag. This is however decided by the value of the secret `b`. Similarly, for `constant_time_2`, the `jne` jump at line 27 depends on the boolean `b` and the code is thus not constant-time. The code for `constant_time_2` is interesting as the compiler manages to optimize away the `&` operator and only uses XOR operations. In the case when `b` is false, the instruction at line 28 sets `eax` to zero as the compiler managed to conclude that the `((y ^x) & (-(unsigned) b))` operation would result in zero. `eax` is then XORed with `ecx` which contains variable `x`. In the other branch, the operation at line 32 moves `y` into `eax`, then stores the result of `y ^x` into `eax` at line 33. The AND operation was removed as it is redundant. However, a peephole optimization could have noticed that the operations at line 33 and 34 are redundant, as the result in `eax` is the same before and after the two operations.

One could argue that it is not really harmful as both branches contain the exact same number of operations for the compiled `constant_time_1` function. However, this does not protect the program from an attack. For instance, an attacker could manage to modify the cache so that the `leal` load instruction is faster in one of the branches. This would make an attacker be able to distinguish which branch was taken and thus leak the secret.

What's most worrying is that `constant_time_2` uses the style of code recommended by cryptographers[4] that abuses bitwise operators in the hope that compilers do not manage to optimize it and therefore not break constant-time security.

Another example can be found in [8] where the authors present a timing attack on a constant-time implementation of an elliptic curve by exploiting the MSVC compiler which transforms a constant-time 64-bit multiplication into a variable-time routine on architectures that do not natively support 64-bit integers.

In order to tackle this issue, we present how to derive a natural methodology to prove preservation of cryptographic constant-time security by adapting the simulation theory that is usually used in order to prove compiler correctness. More specifically, we will give insights on the issues that need to be tackled in order to adapt this framework to the formally verified CompCert C compiler [10].

This paper is organized as follows. First, section II recalls background on the CompCert compiler. Section III derives a theoretical framework that can be used to prove that a correct compiler preserves constant-time security by taking advantage of its proofs of correctness. Section IV presents some insights on why we think CompCert preserves constant-time security and our experience trying to apply the framework we present. Finally, section V presents some related work and details the

[3]Tested on March 1st, 2018 using the Godbolt compiler explorer https://godbolt.org/g/dx4nzC

[4]For example, it is recommended in page 9 of RFC7748 Elliptic Curves for Security https://tools.ietf.org/html/rfc7748#page-10

main differences with [3] which presents a work concurrent to ours with a similar approach while section VI concludes.

II. CompCert

CompCert is an optimizing compiler that compiles C programs down to assembly code. It has 20 compilation passes bridging the gap between 11 intermediate languages from C down to assembly. Each intermediate language is defined by a formal semantics that associates to each program to observable behaviors. Observable behaviors can be normal termination, abnormal termination (due to a runtime error such as division by zero for instance) or divergence. Divergence can be silent if it computes something forever for instance, or non-silent if the user can also observe input/output such as writing to the terminal.

A seemingly desirable property for a compiler is that its generated code has the same observable behavior as the given source code. However, there are two considerations that are not taken into account. First, assembly is a deterministic language while the source language may not be. For instance, the C standard allows for several evaluation order of expressions. Consequently, compilers usually pick one specific evalation order. Second, this does not take into account one of the many pitfalls of C which is undefined behavior. The compiler can replace undefined behavior by any of its choice.

CompCert thus comes with the following semantics preservation theorem:

Theorem 1 (Semantics preservation)**.** If the compiler transforms source code S into compiled code C, without reporting errors, then every observable behavior of C is an observable behavior of S, or it "improves" over one of C by replacing undefined behaviors.

The semantics preservation theorem is actually a corollary of another property called simulation diagrams. Each transformation pass is associated with a simulation diagram and these diagrams are then composed together to establish a diagram for the whole compiler from which the semantics preservation theorem is derived. Simulation diagrams allows us to reason "locally" while proving a property "global" to a program, they form the crux of the methodology we present in the following section.

III. Framework

Suppose that we have a compiler that compiles programs in a source language $\mathcal{S}$ to a target language $\mathcal{T}$ modeled by a partial function $\mathsf{compile} : \mathcal{S} \rightharpoonup \mathcal{T}$. We further assume that both languages are deterministic as it will make further reasoning easier and that the compiler is correct, i.e., it satisfies the following theorem:

Theorem 2 (Correctness of compilation)**.** For all source program p, if p is safe and compiles into program $\mathsf{compile}(p) = p'$, then p' has the same observable behavior as p.

As before, "safe" means no undefined behavior, the semantics of the program does not get stuck. Observable behavior corresponds to the trace of events that can be observed when executing the program, such as asking an input to an user on the command line or writing an integer to it. Whether the program terminates can also be observed.

The theorem only states that observable behavior is preserved, it has no relation with constant-time security. Therefore, a correct compiler does not give guarantee that security is preserved.

We assume that a program has a unique initial state that is determined by the initial values contained in the program. In C and in CompCert, this is determined by the `main` function and all global declarations, i.e., the global variables and the function definitions. A program may have no initial state if it is not well-formed, for instance if it does not contain a `main` function. Having a unique initial state allows to state constant-time security informally as if two programs are "similar" then they have "same leakage". We will use a predicate $\phi(p, p')$ to say that both programs have the same values for some initial public variables that are defined by ϕ and that both the programs are syntactically equal otherwise. It reads as p and p' are ϕ-similar. Given a smallstep semantics with transition $\cdot \to \cdot$, we use $s \xrightarrow{l} s'$ to say that the semantic step from state s to state s' produces the leak l. For constant-time security, the leak is either the value of a branching condition, i.e., executing `if (x) ...` leaks the value of `x`, or the address of a memory access, i.e., `*p = *x + 5` leaks the value of `p` and `x`.

As we previously assumed the languages to be deterministic, constant-time security can thus be defined as follows:

Definition 1 (Constant-time security)**.** A program p is ϕ-constant-time if for any program p' such that $\phi(p, p')$ then p and p' have same leakage, i.e., if s_0 and s'_0 are the initial states of respectively p and p', then for all $n \in \mathbb{N}$, s_1 and s'_1 such that $s_0 \to^n s_1$ and $s'_0 \to^n s'_1$, then either there exists a (possibly empty) leak l, s_2 and s'_2 such that $s_1 \xrightarrow{l} s_2$ and $s'_1 \xrightarrow{l} s'_2$ or both executions are stuck at s_1 and s'_1.

Constant-time security can be stated as a non-interference property as previously, but it can also be defined with a simulation-based view. This will be more useful as all compiler correctness proof are usually stated as a simulation, and thus constant-time security preservation amounts to proving that simulations can be composed in a certain way that we will detail later. Without determinacy, this property wouldn't be "strong" enough as it uses an existential quantifier which does not constrain the actual executions of the programs to follow the execution given by the quantifier.

In order to prove that a program p is ϕ-constant-time, it suffices to prove that p is safe and that for all program p' such that $\phi(p, p')$, there exists a leak-preserving lockstep simulation illustrated in Figure 1 and defined as follows:

Definition 2 (Leak-preserving lockstep simulation)**.** A leak-preserving lockstep simulation between a program p and a program p' is defined by a relation $\cdot \sim \cdot$ between states of p and states of p' such that:

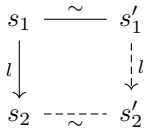


Figure 1: Leak-preserving lockstep diagram
(Hypotheses in plain lines, conclusion in dashed lines)

- If s_i is the initial state of p and s'_i is the initial state of p', then $s_i \sim s'_i$;
- For every step $s_1 \xrightarrow{l} s_2$ leaking information l of program p and state s'_1 of p' such that $s_1 \sim s'_1$, there exists a state s'_2 such that $s'_1 \xrightarrow{l} s'_2$ and $s_2 \sim s'_2$;
- For every state s and s' such that $s \sim s'$, if s is a final state, then so is s'.

Given a leak-preserving lockstep simulation, we prove that it implies same leakage in the following lemma.

Lemma 1. If p is safe and there is a leak-preserving lockstep simulation $\cdot \sim \cdot$ between p and p', then they have same leakage.

Proof. Both p and p' have an initial state, respectively s_0 and s'_0.

We first prove by induction on $n \in \mathbb{N}$ that if $s_0 \rightarrow^n s_n$ and $s'_0 \rightarrow^n s'_n$ then $s_n \sim s'_n$.

- For $n = 0$, we only need to prove that $s_0 \sim s'_0$ which is true by definition of a leak-preserving lockstep simulation;
- Let's now prove for $n+1$ assuming that it is true for n. We have $s_0 \rightarrow^n s_n \rightarrow s_{n+1}$ and $s'_0 \rightarrow^n s'_n \rightarrow s'_{n+1}$. By induction hypothesis, we know that $s_n \sim s'_n$. Thus, by using the leak-preserving lockstep simulation, we have that there exists s''_{n+1} such that $s'_n \rightarrow s''_{n+1}$and $s_{n+1} \sim s''_{n+1}$ (we omit the leak given by the simulation as we don't need it). However, since we assume the languages deterministic, we have that $s'_{n+1} = s''_{n+1}$, and thus, $s_{n+1} \sim s'_{n+1}$.

The property is thus proven by induction.

Now, we prove that both programs have same leakage, i.e., for all $n \in \mathbb{N}$, if $s_0 \rightarrow^n s_n$ and $s'_0 \rightarrow^n s'_n$, then either both executions are stuck at s_n and s'_n, or there exists a leak l_n and states s_{n+1} and s'_{n+1} such that $s_n \xrightarrow{l_n} s_{n+1}$ and $s'_n \xrightarrow{l_n} s'_{n+1}$.

This is true since for any such s_n and s'_n, we just proved that $s_n \sim s'_n$. And since we assume that p is safe, either s_n is a final state of p and therefore s'_n is also a final state of p' thanks to the simulation, or there exists a leak l_n and a state s_{n+1} such that $s_n \xrightarrow{l_n} s_{n+1}$, and again, by the leak-preserving lockstep simulation and by determinacy, there exists a unique s'_{n+1} such that $s'_n \xrightarrow{l_n} s'_{n+1}$.

Finally, we proved that both programs have same leakage. □

However, the converse is not generally true, if two programs have the same leakage, it does not mean that either of them is safe. It is not a problem as we assume a compiler correctness setting, i.e., we assume that the source program is safe. The following lemma can thus be considered the converse of the previous one.

Lemma 2. If p and p' have same leakage, then there exists a leak-preserving lockstep simulation between p and p'.

Proof. Let s_0 and s'_0 be the initial states of respectively p and p'. We define $s \sim s'$ as there exists $n \in \mathbb{N}$ such that $s_0 \rightarrow^n s$ and $s'_0 \rightarrow^n s'$.

- We have trivially $s_0 \sim s'_0$ by taking $n = 0$.
- If $s_1 \xrightarrow{l} s_2$ and $s_1 \sim s'_1$, we need to prove that there exists s'_2 such that $s'_1 \xrightarrow{l} s'_2$ and $s_2 \sim s'_2$. Such a s'_2 exists, since by definition of $s_1 \sim s'_1$, there exists a n such that $s_0 \rightarrow^n s_1$ and $s'_0 \rightarrow^n s'_1$. Since $s_1 \xrightarrow{l} s_2$, there exists s'_2 such that $s'_1 \xrightarrow{l} s'_2$ or p and p' wouldn't have same leakage. Furthermore, $s_2 \sim s'_2$ by definition.
- If s is the final state of p and $s \sim s'$, then s' is the final state of p', or there would be l and s'' such that $s' \xrightarrow{l} s''$ which is impossible since p and p' have same leakage.

The leak-preserving lockstep simulation is thus defined. □

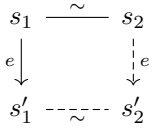
Constant-time security is a symmetrical property in the sense that if p and p' have same leakage, then p' and p have same leakage. Thus, an equivalent definition would be that there exists a leak-preserving lockstep simulation between p and p' and another one between p' and p. However, we chose to trade the second simulation with the assumption that p is safe. This trade has a few advantages, in that we only need to prove one simulation instead of two to prove that a program is constant-time. Furthermore, assuming that the program given to the compiler is safe is a reasonable assumption that is also made when proving the correctness of the compiler.

We have shown that constant-time security implies existence of leak-preserving lockstep simulations, while safety and lockstep simulations are needed to prove constant-time security. Therefore, one possible way to prove the preservation of constant-time security through compilation is to 1. prove that safety is preserved through compilation, 2. the leak-preserving lockstep simulations are preserved through compilation and 3. assume that the initial program is safe. Preservation of safety is already a consequence of the correctness of the compiler.

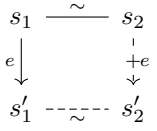
We now have to solve the issue of how to preserve leak-preserving lockstep simulations. Compiler correctness can be stated as trace preservation and is proven through the usage of events-preserving simulations. There are several kinds of such simulations, from the most constrained to the most general, they are the lockstep, plus and star simulations illustrated in Figure 2. We remind the definition of the star simulation as it is the most general one.

Definition 3 (Event preserving star simulation)**.** An event preserving star simulation between a program p and a program p' is defined by a relation $\cdot \sim \cdot$ between states of p and states of p' such that:

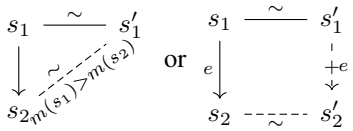
- If s_i is the initial state of p and s'_i is the initial state of p', then $s_i \sim s'_i$;



(a) Lockstep diagram



(b) Plus diagram



(c) Star diagram

Figure 2: Trace preserving simulations
(Hypotheses in plain lines, conclusion in dashed lines)

- There exists a measure function $m : \mathbb{S} \to \mathbb{N}$ where $\mathbb{S}$ is the type of states of p;
- For every step $s_1 \xrightarrow{e} s_2$ producing event e of program p and state s_1' of p' such that $s_1 \sim s_1'$, either there exists a state s_2' such that $s_1' \xrightarrow{e}{}^{+} s_2'$ and $s_2 \sim s_2'$, or e is a silent event (i.e., the step produces no event) and $m(s_1') < m(s_1)$;
- For every state s and s' such that $s \sim s'$, if s is a final state, then so is s'.

The measure function used in the star simulation is to prevent p' from stuttering. Otherwise, a non-terminating program can be compiled into a terminating program and thus violates observable behavior preservation. For instance, suppose the source program is an infinite loop that does nothing, and it is compiled into a single instruction `skip`. Without the measure, the star simulation could be proven, even though behavior has not been preserved, since the source program is non-terminating while the compiled program is terminating.

Intuitively, we can see that the lockstep simulation used for constant-time security and the simulations used for compiler correctness can be composed. Suppose that we have two source programs p and p' such that p is ϕ-constant-time and $\phi(p, p')$. The leak-preserving lockstep simulation $\sim_S$ (S as in Source) tells us that if s_1 is a state of p and s_1' is a state of p' such that $s_1 \sim_S s_1'$ and s_1 advances to some state s_2 while leaking l, i.e., $s_1 \xrightarrow{l} s_2$, then there exists a state s_2' such that $s_1' \xrightarrow{l} s_2'$ and $s_1' \sim_S s_2'$. As we assume the compiler is correct, we know that there is some simulation $\sim_C$ (C as in Compile) to prove that p is correctly compiled, and similarly a simulation $\sim_C'$ for p'. The first simulation tells us that since $s_1 \xrightarrow{l} s_2$, for all σ_1 such that $s_1 \sim_C \sigma_1$, there exists a leak λ and a state σ_2 such that $\sigma_1 \xrightarrow{\lambda}{}^{n} \sigma_2$ where n is some unknown integer. Similarly for the second simulation, it tells us that since $s_1' \xrightarrow{l} s_2'$, for all σ_1' such that $s_1' \sim_C' \sigma_1'$, there exists a leak λ' and a state σ_2' such that $\sigma_1' \xrightarrow{\lambda'}{}^{n'} \sigma_2'$ where n' is some unknown integer.

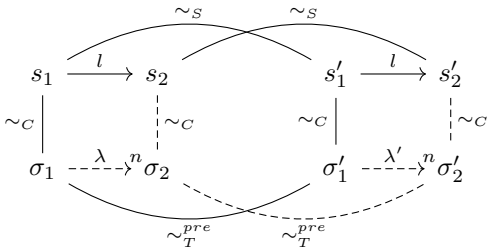


Figure 3: 2-simulation diagram
(Hypotheses in plain lines, conclusion in dashed lines)

This feels like the beginning of a simulation diagram, but still requires proving that $\lambda = \lambda'$ and $n = n'$. We do not need to prove that $\lambda = l$ as leaks are generally not preserved by compilation. For instance, some optimization may remove memory accesses if it deems them unnecessary, the leak due to the memory accesses at the source level is thus removed when compiled. What's important is that the compiled leaks stay the same, i.e., $\lambda = \lambda'$.

We define this as a 2-simulation diagram that is characterized by three relations $(\sim_S, \sim_T^{pre}, \sim_C)$ and detailed below. The last relation $\sim_C$ corresponds to the relation used in proving that the source program is correctly compiled into the target program. There should be two such relations since there are two programs p and p', however, these two relations are morally the same as both programs have been compiled with the same transformation. We thus use only one relation $\sim_C$ for the sake of readability.

Definition 4 (2-simulation diagram)**.** $(\sim_S, \sim_T^{pre}, \sim_C)$ is a 2-simulation diagram for programs p, p', ρ, ρ' if

- $\sim_S$ is a leak-preserving lockstep simulation at source level between p and p',
- $\sim_C$ is an event preserving star simulation between p and ρ that proves the correctness of compiling p into ρ,
- $\sim_C$ is an event preserving star simulation between p' and ρ' that proves the correctness of compiling p' into ρ',

and $\sim_T^{pre}$ is a target level relation between states of ρ and ρ' such that

- if σ_0 and σ_0' are respectively the initial states of ρ and ρ', then $\sigma_0 \sim_T^{pre} \sigma_0'$,
- for all states s_1, s_1', s_2, s_2', σ_1, σ_1' and leak l such that $s_1 \sim_S s_1'$, $s_1 \sim_C \sigma_1$, $s_1' \sim_C \sigma_1'$, $\sigma_1 \sim_T^{pre} \sigma_1'$, $s_1 \xrightarrow{l} s_2$, $s_1' \xrightarrow{l} s_2'$, then there exists an integer n, a leak λ and states σ_2, σ_2' such that $\sigma_1 \xrightarrow{\lambda}{}^{n} \sigma_2$, $\sigma_1' \xrightarrow{\lambda}{}^{n} \sigma_2'$, $s_2 \sim_C \sigma_2$, $s_2' \sim_C \sigma_2'$ and $\sigma_2 \sim_T^{pre} \sigma_2'$, this is illustrated in Figure 3
- for all states σ and σ', if $\sigma \sim_T^{pre} \sigma'$ and σ is a final state, then so is σ'.

Informally, the relation $\sim_T^{pre}$ defined in the 2-simulation diagram represents the fact that the two programs are at the exact same program point. How to define this is however

dependent on the language, which is why we cannot abstract it away in the definition. Furthermore, the relation may not be a leak-preserving lockstep simulation relation as the diagram only tells us that there is some number of steps n between states that are related by $\sim_T^{pre}$, we are missing the lockstep part of the definition. We can however use it to build such a relation as proven by the following theorem, hence the pre in the symbol, as it can be seen as a pre-lockstep simulation.

We only consider program transformations that do not depend on the secrets. For instance, a transformation that would add n `skip` instructions at the beginning of the program is not allowed if n is secret. This is necessary in order to have transformations that verify the property that if p is compiled into ρ and ρ and ρ' are ϕ-similar, then there exists p' such that p and p' are ϕ-similar.

Theorem 3 (Preservation of constant-time security)**.** If $\mathsf{compile}$ does not depend on secrets and program p is ϕ-constant-time, safe and there is a $(\sim_S, \sim_T^{pre}, \sim_C)$ 2-simulation diagram for all p' such that $\phi(p, p')$, then $\mathsf{compile}(p)$ is ϕ-constant-time.

Proof. Let ρ' be a program such that $\phi(\mathsf{compile}(p), \rho')$, there exists a p' such that $\rho' = \mathsf{compile}(p')$ by hypothesis as explained just before the theorem. We first define the relation $\cdot \sim_T^n \cdot$ between states of $\mathsf{compile}(p)$ and $\mathsf{compile}(p')$ as follows:

$$\sigma \sim_T^n \sigma' \triangleq \exists \lambda, \exists \sigma_1, \exists \sigma_1', \exists s_1, \exists s_1', \sigma \xrightarrow{\lambda}{}^n \sigma_1 \wedge \sigma' \xrightarrow{\lambda}{}^n \sigma_1' \wedge \sigma_1 \sim_T^{pre} \sigma_1' \wedge s_1 \sim_C \sigma_1 \wedge s_1' \sim_C \sigma_1' \wedge s_1 \sim_S s_1'.$$

We now define the lockstep simulation relation $\cdot \sim_T \cdot$ between states of $\mathsf{compile}(p)$ and $\mathsf{compile}(p')$ as $\sigma \sim_T \sigma' \triangleq \exists n, \sigma \sim_T^n \sigma'$.

Informally, this means that $\sigma \sim_T \sigma'$ if there exists some states σ_1 and σ_1' such that σ and σ' can both respectively reach σ_1 and σ_1' in the same number of steps while leaking the same information. Furthermore, there must exist some states s_1 and s_1' in the source programs such that $s_1 \sim_C \sigma_1$ and $s_1' \sim_C \sigma_1'$ and $s_1 \sim_S s_1'$.

We first show a lemma that for all n, σ_1 and σ_1', if $n > 0$ and $\sigma_1 \sim_T^n \sigma_1'$, there exists λ, σ_2 and σ_2' such that $\sigma_1 \xrightarrow{\lambda} \sigma_2$, $\sigma_1' \xrightarrow{\lambda} \sigma_2'$ and $\sigma_2 \sim_T^{n-1} \sigma_2'$.

By definition of $\sim_T^n$, there exists λ, σ_3, σ_3', s_3, s_3' such that $\sigma_1 \xrightarrow{\lambda}{}^n \sigma_3$, $\sigma_1' \xrightarrow{\lambda}{}^n \sigma_3'$, $\sigma_3 \sim_T^{pre} \sigma_3'$, $s_3 \sim_c \sigma_3$, $s_3' \sim_C \sigma_3'$ and $s_3 \sim_S s_3'$. Thus, there exists σ_2, σ_2', λ_1 and λ_2 such that $\sigma_1 \xrightarrow{\lambda_1} \sigma_2 \xrightarrow{\lambda_2}{}^{n-1} \sigma_3$, $\sigma_1' \xrightarrow{\lambda_1} \sigma_2' \xrightarrow{\lambda_2}{}^{n-1} \sigma_3'$ and $\lambda = \lambda_1 \cdot \lambda_2$. Hence, we can conclude that $\sigma_2 \sim_T^{n-1} \sigma_2'$ by definition.

We now show that $\cdot \sim_T \cdot$ is indeed a lockstep simulation:

- If σ_i is an initial state of $\mathsf{compile}(p)$ and σ_i' is the initial state of $\mathsf{compile}(p')$, by safety of p and p', there exists s_i and s_i' respectively initial states of p and p'. By definition of $\sim_C$, we have $s_i \sim_C \sigma_i$ and $s_i' \sim_C \sigma_i'$, thus $\sigma_i \sim_T \sigma_i'$ with $n = 0$.
- If σ_f is a final state and $\sigma_f \sim_T \sigma_f'$, by definition of $\sim_T$, there exists some states s and s' such that $s \sim_C \sigma_f$, $s' \sim_C \sigma_f'$ and $s \sim_S s'$.
  By definition of a star simulation and since σ_f is a final state, there exists a state s^1 such that $s \to s^1$, $s^1 \sim_C \sigma_f$ and $m(s^1) < m(s)$. By iterating this process, we build a finite maximal sequence $s^1, \ldots, s^k$ of states such that $s \to s^1 \to \ldots \to s^k$ and $s^k \sim_C \sigma_f$. The sequence is finite because we have $m(s^1) > \ldots > m(s^k)$ and this cannot infinitely decrease as $\mathbb{N}$ is well-founded. $s_f = s^k$ is a final state, since otherwise there would be a state s^{k+1} such that $s^k \to s^{k+1}$ and the sequence wouldn't be maximal.
  And by exploiting the lockstep simulation $\sim_S$, we can build a sequence of states $s'^1, \ldots, s'^k$ such that $s' \to s'^1 \ldots \to s'^k$, $s^1 \sim_S s'^1, \ldots, s^k \sim_S s'^k$. Since $s_f = s^k$ is a final state, then so is $s_f' = s'^k$ thanks to the lockstep simulation $\sim_S$.
  By definition of the 2-simulation diagram, $s_f' \sim_C \sigma_f'$, thus σ_f' is also a final state.
- If $\sigma_1 \sim_T \sigma_1'$ and $\sigma_1 \xrightarrow{\lambda} \sigma_2$, by definition of $\sim_T$, there exists n, σ_3, σ_3', λ' such that $\sigma_1 \xrightarrow{\lambda'}{}^n \sigma_3$ and $\sigma_1' \xrightarrow{\lambda'}{}^n \sigma_3'$ and there exists s, s' such that $s \sim_C \sigma_3$, $s' \sim_C \sigma_3'$ and $s \sim_S s'$. We need to prove there exists σ_2' such that $\sigma_1' \xrightarrow{\lambda} \sigma_2'$ and $\sigma_2 \sim_T \sigma_2'$.
  - If $n > 0$, we use the lemma, and therefore there exists λ_{bis}, σ_{2bis}, σ_2' such that $\sigma_1 \xrightarrow{\lambda_{bis}} \sigma_{2bis}$, $\sigma_1' \xrightarrow{\lambda_{bis}} \sigma_2'$ and $\sigma_{2bis} \sim_T \sigma_2'$. By determinism of the semantics, we have that $\sigma_2 = \sigma_{2bis}$ and $\lambda = \lambda_{bis}$. Thus we have $\sigma_1' \xrightarrow{\lambda} \sigma_2'$ and $\sigma_2 \sim_T \sigma_2'$.
  - However, if $n = 0$, we have $\sigma_3 = \sigma_1$ and $\sigma_3' = \sigma_1'$. Thus, we obtain $s \sim_C \sigma_1$ and $s' \sim_C \sigma_1'$. s cannot be a final state, because σ_1 would be a final state due to $\sim_C$ which is impossible since $\sigma_1 \xrightarrow{\lambda} \sigma_2$. Hence, by safety of p, there exists a state s_2 such that $s \xrightarrow{l} s_2$. By the definition of lockstep simulation with $\sim_S$, there exists some s_2' such that $s' \xrightarrow{l} s_2'$ and $s_2 \sim_S s_2'$. By using the 2-simulation diagram, there exists $k \in \mathbb{N}$, σ_4, σ_4' and λ^1 such that $\sigma_1 \xrightarrow{\lambda^1}{}^k \sigma_4$ and $\sigma_1' \xrightarrow{\lambda^1}{}^k \sigma_4'$ with $s_2 \sim_C \sigma_4$, $s_2' \sim_C \sigma_4'$ and $\sigma_4 \sim_T^{pre} \sigma_4'$. Therefore, by definition, $\sigma_1 \sim_T^k \sigma_1'$.
    If $k > 0$, we use again the previous lemma to conclude.
    Otherwise $k = 0$, and we know that $m(s_2) < m(s)$ by definition of $\sim_C$. Thus, we can reiterate the previous process until we obtain a new "k" that is strictly positive. This iteration process is finite because the measure strictly decrease until we obtain such a new k and it cannot decrease infinitely. The conclusion is hence the same as before.

We proved that $\sim_T$ is a lockstep simulation, thus $\mathsf{compile}(p)$ is ϕ-constant-time thanks to Lemma 1 and the theorem is proven. □

We proved that if the 2-simulation diagram is satisfied, then constant-time security is preserved. However, it is still left to prove that the simulation diagram can be satisfied by a

compiler. Intuitively, we only know that given a star simulation $\sim_C$, when the states of the two high level programs advance, the lower level states will advance some number of steps n and n' which are not necessarily equal. However, the high level programs are in a lockstep simulation and thus follow *a fortiori* the same control flow, it makes sense that the lower level states advance similarly. We now illustrate how to instantiate the framework in CompCert.

IV. Application to CompCert

We study in this section how the method presented previously can be adapted to CompCert. We first need to define our models by first defining what it means for programs to be similar, then what are the leaks we consider and finally how to augment each semantics with leaks. We will then review the different compilation passes of CompCert.

A. Adapting the framework

In CompCert, a program is represented by the identifier of its `main` and a list of declarations which are global variables and function definitions. Thus, we can define similarity of programs `p1` and `p2` with regard to a set of identifiers that represent secret variables as `p1` and `p2` have the same `main` identifiers and the same function definitions, global variables are only allowed to differ if their identifiers are in the set of secret variables and are otherwise equal. This can be defined as follows in Coq where `match_except secret` is a predicate that says that the program definitions are similar except for variables in `secret` and `list_forall2 p l1 l2` means that for every element $a_1, a_2, \ldots$ of `l1` and $b_1, b_2, \ldots$ of `l2`, `p` a_i b_i holds.

```
Definition similar_programs
  (secret: list ident)
  (p1 p2: program): Prop :=
    p1.(prog_main) = p2.(prog_main) /\
    list_forall2 (match_except secret)
                 p1.(prog_defs)
                 p2.(prog_defs)
```

We then need to instantiate our model of leaks. For constant-time security, the leaks are either `Guard b` where `b` is a boolean due to the evaluation of the guard clause in a conditional, a memory access `MemAccess block ptrofs` or the leak is `Silent`.

Finally, in order for leaks to appear in semantics, we can rewrite each semantics to incorporate them but this would require extensive changes at all levels of the compiler. A more modular way is to define an observation predicate `observe` for each semantics and define a "leaky" step as

```
Definition lstep (sem: semantics)
                 (observe: state sem ->
                           leak -> Prop)
                 (s1: state sem) (l: leak)
                 (s2: state sem) :=
                   exists e, step sem s1 e s2 /\
                   observe s1 l.
```

`observe s1 l` means that when advancing from state `s1`, `l` will be leaked. `s2` is not needed as the leak is entirely determined by what's executed which is contained in `s1`. We can now state constant-time security.

```
Definition secure (secret: list ident)
                  (p: program): Prop :=
  forall (p': program),
   similar_programs secret p p' ->
    forall s0 s0',
     initial_state (sem p) s0 ->
     initial_state (sem p') s0' ->
      forall n s1 s1' t t',
       StarN (semantics p) n s0 t s1 ->
       StarN (semantics p') n s0' t' s1' ->
        (exists l s2 s2',
          lstep (sem p) observe s1 l s2 /\
          lstep (sem p') observe s1' l s2') \/
           (~ exists e s2,
              step (semantics p) s1 e s2 /\
            ~ exists e' s2',
              step (semantics p') s1' e' s2').
```

This is exactly Definition 1 written in Coq, a program `p` is secure if for all programs `p'` that are similar with `p` with regards to `secret`, then if `s0` and `s0'` are respectively their initial states, then for all states `s1` and `s1'` such that `s0` $\rightarrow^n$ `s1` and `s0'` $\rightarrow^n$ `s1'`, either both states `s1` and `s1'` can take a leaky step with same leak `l`, or both executions are stuck.

A leak-preserving lockstep simulation is defined as a record in Coq.

```
Record lp_sim_properties
  (match_states: state -> state -> Prop)
  : Prop :=
  Build_lp_sim_properties {
      lp_match_initial_states:
        forall s1,
          initial_state sem1 s1 ->
          exists s2, initial_state sem2 s2
                     /\ match_states s1 s2;
      lp_match_final_states:
        forall s1 s2 r,
          match_states s1 s2 ->
          final_state sem1 s1 r ->
          final_state sem2 s2 r;
      lp_simulation: forall s1 l s1',
          lstep sem1 s1 l s1' ->
          forall s2,
            match_states s1 s2 ->
            exists s2',
              lstep sem2 s2 l s2' /\
              match_states s1' s2' }.
```

The definition in Coq follows exactly Definition 2 but renames the $\sim$ relation into `match_states`.

The next step is to define the framework for 2-simulations. However, its definition relies on stating that the two executions at the target level (bottom part of Figure 3) advance the *same* number of steps. This number of steps is not random but is the number of steps prescribed by the event preserving simulation used for proving the correctness of the compiler. Yet, this number of steps does not appear explicitly in the theorem statement in CompCert as shown below.

```
fsim_simulation:
      forall s1 t s1', Step L1 s1 t s1' ->
      forall i s2, match_states i s1 s2 ->
      exists i', exists s2',
        (Plus L2 s2 t s2' \/
        (Star L2 s2 t s2' /\ order i' i))
      /\ match_states i' s1' s2'.
```

This proposition states that if a state `s1` of semantics `L1` advances to `s1'` while producing event `t` and it is related with state `s2` such that `match_states i s1 s2`, then there exists an index `i'` and a state `s2'` such that `s2` advances to `s2'` while producing event `t` and `s1'` and `s2'` are related, if the number of steps is not strictly positive (`Star` case), then `i'` must be less than `i` (i.e., `order i' i`); the indexes `i` and `i'` represent the decreasing measure that we used in the previous section.

The number of steps does not appear at all, but it is a crucial part of our framework. Furthermore, we cannot only just state that there *exists* some number of steps as it would then be impossible to relate it to the number of steps taken by the "second" execution and make it impossible to reason with. One observation that can be made is that this number of steps already appears in the *proof* of the statement as the steps taken by `s2` are described inside of the proof. Moreover, as this number of steps only depends on how are `s1` and `s2` related, i.e., `match_states i s1 s2`, the simulation statement can be amended this way into a "counting" simulation.

```
counting_fsim_simulation:
      forall s1 t s1', Step L1 s1 t s1' ->
      forall n i s2,
        match_states n i s1 s2 ->
        exists s2', exists i', exists n',
          (StarN L2 n s2 t s2' /\
          (n = 0 -> order i i'))
          /\ match_states n' i' s1' s2'.
```

The `match_states` relation is modified in order to take an additional parameter `n` which is a natural number that represents the number of steps taken by `s2` to reach `s2'`, if `n` is zero then the index must decrease. From our experiments on a few passes in CompCert, the necessary modifications to the proofs seem fairly minor.

Finally, we can state the 2-simulation diagram in Coq.

```
simulation_diagram:
  forall s1 s2 s1' s2' l,
    lstep sem1 obs s1 l s2 ->
    lstep sem2 obs s1' l s2' ->
    match_statesS s1 s1' ->
    match_statesS s2 s2' ->
    forall n i i' sigma1 sigma1',
      match_statesC n i s1 sigma1 ->
      match_statesC n i' s1' sigma1' ->
      match_statesPreT sigma1 sigma1' ->
      exists n' i2 i2' l2 sigma2 sigma2',
      Nlstep sem1' obs' n sigma1 l2 sigma2 /\
      Nlstep sem2' obs' n sigma1' l2 sigma2' /\
      match_statesC n' i2 s2 sigma2 /\
      match_statesC n' i2' s2' sigma2' /\
      match_statesPreT sigma2 sigma2'.
```

This is the Coq definition of the property illustrated in Figure 3 with a minor change in that we assume that the two target executions will both advance the same number of steps `n` provided by the counting simulation, instead of proving there exists such a `n`. This does not modify the property conceptually.

We now give a quick review of the different compilation passes of CompCert.

B. Analysis of the Compilation Passes

We now review the different transformation passes in CompCert and try to explain what issue each one could bring or give insights on why they preserve constant-time security.

The first compilation pass of CompCert is named `SimplExpr`, its purpose is to pull side-effects out of expressions, for instance `x = 2 + y++` can be transformed into `tmp = y; y = y + 1; x = 2 + tmp` where `tmp` is a new variable that is introduced by the transformation. This pass preserves constant-time security as nothing is changed during compilation except for making explicit the order of evaluation.

The second pass is named `SimplLocals`, its purpose is to pull scalar local variables out of memory. In CompCert C, all variables live in memory, however, starting from the second intermediate language, Clight, some variables can live in registers. Clight is the language on which `SimplLocals` operates. The meaning of scalar local variables is those variables local to a function that the compiler can statically determinates that their addresses are never used, and can thus decide to safely put them into registers. Consequently, this pass may remove memory accesses. However, given two similar source programs, the same variables in both programs will be moved into registers, thus constant-time security is preserved.

The next pass `Cshmgen` replaces the overloaded operators of Clight with explicitly typed operators. This has no impact on memory accesses nor on the control-flow of programs, and the pass thus trivially preserves security.

The Cminorgen pass does the stack allocation, i.e., functions do not allocate memory for each of its local variables, but instead allocate a single *stack* that can hold all of its local variables. For instance, if a function f has a local variable x, then every occurence of &x is replaced with stack_f + ofsx where stack_f is a pointer to the stack of f and ofsx is an integer offset decided during compilation. The offset is constant and thus does depend on secret information, security is hence preserved.

The following pass is named Selection, its purpose is to recognize patterns and to replace them with selected operators specific to the target architecture. Some of these transformations involve changing 64-bits operations into calls to runtime library functions similarly to the issue described in [8]. It is thus necessary to verify that these runtime library functions are not variable-time, which seems to be the case after a cursory manual analysis.

The next pass is RTLgen which does not modify the programs, but only rewrite them in a new intermediate language more prone for optimizations named RTL. Among the optimization passes, Renumber, Deadcode and Unusedglob are different forms of dead code elimination. These passes preserve constant-time security. Indeed, they only remove code that are *never* executed, hence nothing changes between the execution of the source program and the target program's. Inlining is another optimization pass in which designated callee functions are inlined into the caller functions. In order to do that, the stack of the callee function is merged with the stack of the call function. Consequently, leaks due to memory accesses to global variables or local variables of non-inlined functions are unchanged, but those to the local variables of inlined functions are shifted to the stack of the host function at a constant offset, and there is thus not more leaks than before inlining.

The other optimizations are CSE and Constprop which we will detail later.

The next pass is register allocation Allocation. This transformation can introduce "spilling", i.e., variables that should live in registers are put into memory due to lack of registers. They are put on the stack, hence the stack of each function can become bigger. Consequently, this pass may add leaks that correspond to these new memory accesses to spilled variables. However, each spilled variable is put at a constant offset on the stack, hence the memory addresses that are leaked do not depend on secret information. Register allocation preserves security.

We have not yet analyzed the last passes, namely Tunneling, Linearize, CleanupLabels, Debugvar, Stacking and Asmgen but expect them to also preserve security as these passes form the backend of the compiler and should not modify greatly the code.

We now review the Constprop (Constant Propagation) pass in details, the CSE (Common Subexpression Elimination) pass is similar. These passes are different in that they may remove memory accesses instead of only modifying them. For instance, constant propagation can remove a memory load if the analysis manages to prove that it is redundant, x = *p; y = *p can be rewritten into x = *p; y = x.

In order to prove that this pass preserves constant-time security, we need to define the $\cdot \sim_T^{pre} \cdot$ relation presented in the previous section. As explained earlier, $\sigma \sim_T^{pre} \sigma'$ intuitively tells that both states σ and σ' are at the exact same program point. We define this in Coq as an "indistinguishability" relation. We first recall the RTL intermediate language that is used for most optimizations in CompCert such as Constprop.

An execution state in RTL is either a Callstate, a Returnstate or a regular State. They all record a list of stackframes Stackframe res f sp pc rs which contains a caller function f, its corresponding stack pointer sp and the program point where it was left at pc, its register state rs and the register res where the return value must be stored.

A Callstate stk f args m represents a state with the list of stackframes stk and memory m about to call the function f with arguments args. A Returnstate stk v m represents a state with list of stackframes stk and memory m that returns the value v. A State stk f sp pc rs m represents a state with list of stackframes stk, register state rs, memory m, current function f, stack pointer sp and program counter pc.

We define the indistinguishability $\simeq$ for stackframes and states in Figure 4. Two stackframes are indistinguishable if they are equal except for their register states that are allowed to differ. Two states are indistinguishable if their stackframes are indistinguishable and they are at the same program point.

The first property to prove for our 2-simulation is the following one: given programs p, p', ρ and ρ' such that p and p' are respectively transformed into ρ and ρ' after Constprop, the initial states of ρ and ρ' must be indistinguishable. The initial state of a program is Callstate nil f nil m where f is the function corresponding to the main function of the program, the memory m is just initialized with the global variables. Thus, proving that two initial states are indistinguishable comes down to proving that the two main functions are equal as we do not need to prove anything on the memory part. This is trivial as by definition of program similarity, the functions of both ρ and ρ' are pairwise equal, hence their main are equal.

The next step is to fulfill the diagram, part of what needs to be proven is that if two indistinguishable states in the target programs advance the same number of steps, then they both arrive at indistinguishable states. Let's have a closer look to function calls. The semantics for calls at RTL level is defined as follows in CompCert.

```
exec_Icall:
forall s f sp pc rs m sig ros args res
       pc' fd,
       (fn_code f)!pc =
       Some(Icall sig ros args res pc') ->
       find_function ros rs = Some fd ->
       funsig fd = sig ->
       step (State s f sp pc rs m) E0
```

$$\frac{}{\texttt{Stackframe res f sp pc rs} \simeq \texttt{Stackframe res f sp pc rs'}}$$

$$\frac{\texttt{stk} \simeq \texttt{stk'}}{\texttt{State stk f sp pc rs m} \simeq \texttt{State stk' f sp pc rs' m'}}$$

$$\frac{\texttt{stk} \simeq \texttt{stk'}}{\texttt{Callstate stk f args m} \simeq \texttt{Callstate stk' f args' m'}}$$

$$\frac{\texttt{stk} \simeq \texttt{stk'}}{\texttt{Returnstate stk v m} \simeq \texttt{Returnstate stk' v' m'}}$$

Figure 4: Indistinguishability definition

```
(Callstate
  (Stackframe res f sp pc' rs :: s)
  fd rs##args m)
```

The rule says that if the instruction to be executed at program point `pc` is a call instruction `Icall sig ros args res pc'` and that given the register state `rs` and the register or symbol `ros`, the function called is `fd`, then the next state is a `Callstate` about to enter `fd`.

Now, suppose that both indistinguishable states of our target programs are at `Icall` instructions. They thus both arrive at `Callstate`s. In order to prove them indistinguishable, we need to prove that the functions that are called are equal. There are two cases, either they are both called by name, i.e. `ros` is a symbol, or by pointer, i.e. `ros` is a register that contains the pointer value. In the first case, it is easy as both programs are similar, thus the symbol is associated to the same function in both programs. In the second case, it is not that simple. Indeed, we do not know the contents of the register states nor the memory, and cannot thus conclude that both function calls use the same pointer value, and even then we do not know whether the memory layout is different between the two programs.

The first idea one would have is to make use of the fact that in the diagram, there are source states s and s' such that $s \sim_C \sigma$ and $s' \sim_C \sigma'$ in order to exploit the correctness proof of compilation. The proof tells us that the function call in the transformed program corresponds to a call to the transformed form of the function called in the source program. By hypothesis, we know that the two function calls in the source program are equal. We need to be able to deduce from it that the functions called at the target level are equal. This reasoning would work for most passes, but unfortunately not for `Constprop` as it is one of the few program transformations that relies on an external analysis, i.e., the transformation depends on the results of the analysis.

This might not seem a difficult issue, as we could just think that since both programs are similar, then their analyses must be the same. This is true, but it is not that easy in presence of separate compilation which is supported by CompCert. Indeed, a user could compile multiple compilation units separately using CompCert and then link them together afterwards. Thus, the transformation of a function does not depend on the analysis of the *whole* program, but only on the compilation unit that it is in. This is where the issue lies as illustrated below.

```
Lemma functions_translated:
  forall (v: val) (f: fundef),
  Genv.find_funct ge v = Some f ->
  exists cunit,
    Genv.find_funct tge v =
      Some (transf_fundef (romem_for cunit) f)
      /\ linkorder cunit prog.
```

The lemma states that for each function `f` in the initial program (represented by its global environment `ge`), the corresponding function in the transformed program is `transf_fundef (romem_for cunit) f` where `cunit` is a compilation unit contained in the whole program `prog`. This is problematic as the lemma states only that there exists a compilation unit but does not give enough constraint on it in order to relate the two compilation units we obtain from our two target states.

A possible solution is to not use the high-level lemmas provided by CompCert, but use a lower-level reasoning. The solution relies on the way global definitions are allocated during the initialization process. In CompCert, each global variable and function definition is associated a pointer, and this process is determined entirely by the order of the definitions. As we consider our two target programs to be similar, the order of definitions is the same. The global environment (the association table between definitions and pointers in CompCert parlance) is thus the same. Next, the correctness proof tells us that the target program uses the same pointer as in the source program. We thus only have to prove that the two pointers at the source level are the same to prove that they are also the same at the target level. The same pointers are used at the source level because we know that both programs called the same function and thus necessarily used the same pointer.

This shows some of the difficulties besides those inherent to our framework, but are due to the characteristics of adapting to a realistic compiler such as CompCert. Only the definitions given in this section have been formalized in Coq, the proofs presented in III have not been mechanized yet and are left as future work. Furthermore, we presented the troubles we encountered while trying to prove that the constant propagation

pass preserves security, the proof has not been finished yet however.

V. Related Work

A. Verifying constant-time

High assurance cryptography is a flourishing area of research that has spawned many recent projects. Many of them tackle the question of verifying constant-time implementations at either source level [5], assembly [4], or in an intermediate representation [1]. Some propose tools to help write verified implementations of cryptography code. For instance, Vale [6] which proposes a tool to produce verified assembly code that can then be verified for constant-timeness. FaCT [7] proposes a domain-specific language similar to C but with builtin tools that would make it difficult to write non secure code. Jasmin [2] is a formally-verified compiler from the Jasmin language down to assembly. The Jasmin language is a small low-level language similar to Bernstein's qhasm that also supports function calls and high-level control-flow constructs such as loops, thus allowing programmers to easily write correct cryptography code. They also provide tools to verify memory safety and constant-timeness through a sound embedding into Dafny [9]. HACL* [14] is a formally verified C cryptography library derived from code and specifications written in F* [13]. Each implementation is manually verified for functional correctness and constant-timeness.

B. Preservation of constant-time security

Few work formally address the challenge of preserving constant-time security. KreMLin [11] is a compiler from Low* to C that is used to produce HACL*, the authors claim that their compilation preserves constant-time, but we have not managed to understand their proof. Jasmin [2] gives an informal discussion of how their compiler could be proven to preserve constant-time, but leaves it as future work.

In a concurrent work to ours, the approach presented in [3] is in essence very similar to ours. Their paper presents their approach on a While language with a toy compiler built from scratch. This allows them to avoid pitfalls due to design choices of a preexisting compiler such as CompCert. Furthermore, they aim to apply their methodology on the Jasmin compiler [2] which uses a language close to Bernstein's qhasm. This language is already close to assembly, which means that their compiler does not need as many compilation passes as CompCert. Moreover, the compiler being younger may also not have as much technical debt as CompCert, making their endeavor easier.

One notable difference in our methodology is that they require that when `match_states s1 s2` and `s1` advances one step, the number of steps advanced by `s2` must be computable by a function `num_steps` such that the number is `num_steps(s1, s2)`. For their example on the constant propagation pass, this necessitated to enrich the syntax of programs with annotations and thus modify the compilation pass to properly produce these annotations. For instance, in order to define their `num_steps` function, they need

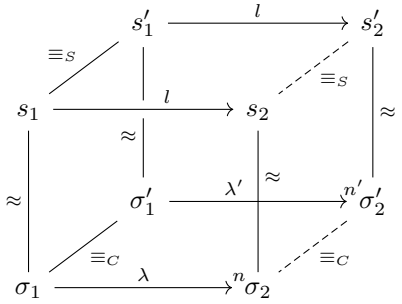


Figure 5: 2-simulation diagram from [3]
(Hypotheses in plain lines, conclusion in dashed lines)

to statically know whether a branch is removed, they have to produce an annotated version of the source program with boolean flags telling whether the branch is removed to accomplish this. Thus, applying their method on CompCert would require to modify the syntax of the language and its semantics, which impacts all compilation passes that uses this language. It is preferable to avoid modifications if possible.

A second difference is that their diagram (illustrated in Figure 5 using their notations) is slightly different from ours (in Figure 3). They directly assume for instance that there exists λ, n and σ_2 such that $\sigma_1 \xrightarrow{\lambda}{}^{n} \sigma_2$ and $s_2 \approx \sigma_2$ where $\approx$ is the relation used in the simulation for proving correctness of the compiler pass, while we ask to prove that such objects exist. They only ask to prove that $\lambda = \lambda'$, $n = n'$ and the dashed lines in the diagram. They are thus asking less things to prove than us. However, it seems intuitive that in order to prove that $\lambda = \lambda'$ in the diagram, it is necessary to be able relate λ and λ' with l. We conjecture that they use the fact that $s_1 \approx \sigma_1$ and determinacy of the semantics in order to relate l and λ for instance. This is similar to unfolding the correctness proof of the transformation in order to relate the source and target leaks which is what our methodology imposes. Thus, in our opinion, the amount of work needed by both methodologies is similar.

VI. Conclusion

In this paper, we showed how to derive a framework for proving preservation of constant-time security from simulation-based compiler correctness proofs. The approach is based on "simulating" 2 executions at once, hence the name of 2-simulations. We presented how this framework could be adapted to the formally verified CompCert C compiler and reviewed its compilation passes to understand which passes may be problematic. We also presented some challenges that occur when trying to prove that the constant propagation pass preserves constant-time security. As the proofs presented in Section III have not been mechanized yet for CompCert, this is the main direction for future work, while finishing the proof of preservation for constant propagation and the other passes are a second direction.

References

[1] José Bacelar Almeida et al. "Verifying Constant-Time Implementations". In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. 2016, pp. 53–70.

[2] José Almeida et al. "Jasmin: High-Assurance and High-Speed Cryptography". In: *CCS 2017-Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017.

[3] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. *Provably secure compilation of side-channel countermeasures*. Cryptology ePrint Archive, Report 2017/1233. https://eprint.iacr.org/2017/1233. 2017.

[4] Gilles Barthe et al. "System-level Non-interference for Constant-time Cryptography". In: *ACM SIGSAC Conference on Computer and Communications Security*. 2014, pp. 1267–1279.

[5] Sandrine Blazy, David Pichardie, and Alix Trieu. "Verifying constant-time implementations by abstract interpretation". In: *European Symposium on Research in Computer Security*. Springer. 2017, pp. 260–277.

[6] Barry Bond et al. "Vale: Verifying high-performance cryptographic assembly code". In: *Proceedings of the USENIX Security Symposium*. 2017.

[7] Sunjay Cauligi et al. "FaCT: A Flexible, Constant-Time Programming Language". In: *Cybersecurity Development (SecDev), 2017 IEEE*. IEEE. 2017, pp. 69–76.

[8] Thierry Kaufmann et al. "When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-donna Built with MSVC 2015". In: *Cryptology and Network Security*. Ed. by Sara Foresti and Giuseppe Persiano. Cham: Springer International Publishing, 2016, pp. 573–582. ISBN: 978-3-319-48965-0.

[9] K Rustan M Leino. "Dafny: An automatic program verifier for functional correctness". In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer. 2010, pp. 348–370.

[10] Xavier Leroy. "A formally verified compiler back-end". In: *Journal of Automated Reasoning* 43.4 (2009), pp. 363–446. URL: http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf.

[11] Jonathan Protzenko et al. "Verified Low-Level Programming Embedded in F*". In: *Proceedings of the ACM on Programming Languages* 1.ICFP (Sept. 2017), 17:1–17:29. DOI: 10.1145/3110261. URL: https://hal.archives-ouvertes.fr/hal-01672706.

[12] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. "Dude, is my code constant time". In: *Proc. of DATE 2017*. 2017.

[13] Nikhil Swamy et al. "Secure distributed programming with value-dependent types". In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming*. Ed. by Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy. ACM, 2011, pp. 266–278. ISBN: 978-1-4503-0865-6. DOI: 10.1145/2034773.2034811. URL: https://www.microsoft.com/en-us/research/publication/secure-distributed-programming-with-value-dependent-types/.

[14] Jean-Karim Zinzindohoué et al. "HACL*: A Verified Modern Cryptographic Library". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: ACM, 2017, pp. 1789–1806. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134043. URL: http://doi.acm.org/10.1145/3133956.3134043.