Formal Verification of a Constant-Time Preserving C Compiler

GILLES BARTHE, MPI for Security and Privacy, Germany and IMDEA Software Institute, Spain
SANDRINE BLAZY, Univ Rennes, Inria, CNRS, IRISA, France
BENJAMIN GRÉGOIRE, Inria, France
RÉMI HUTIN, Univ Rennes, Inria, CNRS, IRISA, France
VINCENT LAPORTE, Inria, France
DAVID PICHARDIE, Univ Rennes, Inria, CNRS, IRISA, France
ALIX TRIEU, Aarhus University, Denmark

Timing side-channels are arguably one of the main sources of vulnerabilities in cryptographic implementations. One effective mitigation against timing side-channels is to write programs that do not perform secret-dependent branches and memory accesses. This mitigation, known as "cryptographic constant-time", is adopted by several popular cryptographic libraries.

This paper focuses on compilation of cryptographic constant-time programs, and more specifically on the following question: is the code generated by a realistic compiler for a constant-time source program itself provably constant-time? Surprisingly, we answer the question positively for a mildly modified version of the CompCert compiler, a formally verified and moderately optimizing compiler for C. Concretely, we modify the CompCert compiler to eliminate sources of potential leakage. Then, we instrument the operational semantics of CompCert intermediate languages so as to be able to capture cryptographic constant-time. Finally, we prove that the modified CompCert compiler preserves constant-time. Our mechanization maximizes reuse of the CompCert correctness proof, through the use of new proof techniques for proving preservation of constant-time. These techniques achieve complementary trade-offs between generality and tractability of proof effort, and are of independent interest.

CCS Concepts: • **Security and privacy** → **Logic and verification**.

Additional Key Words and Phrases: verified compilation, CompCert compiler, timing side-channels

ACM Reference Format:
Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2020. Formal Verification of a Constant-Time Preserving C Compiler. *Proc. ACM Program. Lang.* 4, POPL, Article 7 (January 2020), 30 pages. https://doi.org/10.1145/3371075

1 INTRODUCTION

Formally verified compilers are compilers that come with a machine-checkable proof of correctness. Typically, these correctness proofs show that the compiler preserves the safety and behavior of

Authors' addresses: Gilles Barthe, MPI for Security and Privacy, Bochum, Germany; IMDEA Software Institute, Madrid, Spain, gbarthe@mpi-sp.org; Sandrine Blazy, Univ Rennes, Inria, CNRS, IRISA, Rennes, France, sandrine.blazy@irisa.fr; Benjamin Grégoire, Inria, Sophia-Antipolis, France, benjamin.gregoire@inria.fr; Rémi Hutin, Univ Rennes, Inria, CNRS, IRISA, Rennes, France, remi.hutin@ens-rennes.fr; Vincent Laporte, Inria, Rennes, France, vincent.laporte@inria.fr; David Pichardie, Univ Rennes, Inria, CNRS, IRISA, Rennes, France, david.pichardie@ens-rennes.fr; Alix Trieu, Aarhus University, Aarhus, Denmark, alix.trieu@cs.au.dk.

2475-1421/2020/1-ART7
https://doi.org/10.1145/3371075

programs. Until recently, building formally verified compilers for realistic languages was seen as a grand challenge. However, the advent of the CompCert [Kästner et al. 2018; Leroy 2006, 2009a; Leroy et al. 2016], Vellvm [Zhao et al. 2012, 2013], Crellvm [Kang et al. 2018], and CakeML [Kumar et al. 2014; Lööw et al. 2019; Owens et al. 2017; Tan et al. 2016] verified compilers show that they are now well within reach—although admittedly, every formally verified compiler for a realistic language remains a feat.

Formally verified compilers guarantee the absence of correctness bugs, but do not protect against other classes of bugs, such as performance bugs or security bugs, which also pose serious practical threats. This limitation partly arises from the traditional form of stating compiler correctness as preservation of operational or denotational semantics that do not capture non-functional properties such as performance or security. However, there are some deeper issues, which have led to the emergence of a sub-area of research at the intersection of programming languages, compilation and language-based security. First, there is no one-fits-all definition of secure compilation [Abate et al. 2018; Patrignani et al. 2019]. Second, proof techniques for compiler correctness, including the traditional notions of simulation, do not immediately apply to secure compilation, and need to be extended accordingly. As a consequence, the main focus of secure compilation has remained (understandably so) foundational, and very few works study secure compilation for realistic compilers or even concrete compiler passes.

In this paper, we address the challenges of secure compilation from the specific angle of turning a pre-exisiting formally-verified compiler into a formally-verified secure compiler. This angle of attack is very natural, for three reasons: first, formally-verified compilers for full-fledged languages are major feats, and factoring out this effort is appealing. Second, modular reasoning (in this case first proving correctness then security) is good practice, especially in the context of machine-checked proofs. Third, formally-verified compilers are already adopted, and leveraging their usage seems ingenious. Specifically, we consider the problem of secure compilation for CompCert [Blazy et al. 2006; Leroy 2006, 2009a], a formally-verified moderately optimizing compiler for `C` programs; currently CompCert targets four architectures: `PowerPC`, `ARM`, `x86` and `RISC-V`. The CompCert compiler is programmed (mostly) and verified (fully) using the Coq proof assistant [Inria 2019]. The compiler itself is written as a sequence of 20 compiler passes, from the CompCertC source language down to assembly, going through eight intermediate languages (see Figure 11). Among them are standard optimization passes performing mainly instruction selection, dataflow analysis (used for constant propagation, common subexpression elimination and redundancy elimination), register allocation and function inlining. Compiler passes are either programmed and verified in Coq, or programmed in OCaml and verified in Coq, using translation validation. The correctness theorem states that the compilation is semantics-preserving, i.e., does not introduce bugs in compiled programs. This main theorem decomposes into correctness theorems for each of the 20 compilation passes. Again, each pass decomposes into elementary passes that are proved correct. A general theorem states the correctness of the sequential composition of two passes from the correctness of both passes. CompCert evolved significantly over the last 15 years, starting as an academic project and now being used in commercial settings. For example, CompCert was recently used in the certification of a highly safety-critical application [Kästner et al. 2018]. With 100,000 lines of Coq and 6 person-years of effort, the CompCert proof is among the largest ever performed with a proof assistant.

To make the question "Does the CompCert compiler preserve security?" precise, we must fix the notion of security we want to address. We focus on the specific setting of side-channel protection. More specifically, we consider "cryptographic constant-time", a popular software-based countermeasure against timing-based and cache-based attacks. Informally, an implementation is secure with respect to the cryptographic constant-time policy if its control flow and sequence of memory accesses do not depend on secrets. Thus, cryptographic constant-time is a misnomer and it is perhaps more accurate to think of the cryptographic constant-time policy as the combination of the program counter policy [Molnar et al. 2005] (programs should not branch on secrets) and the memory obliviousness policy [Liu et al. 2013] (memory accesses should not depend on secrets). The practical effectiveness of cryptographic constant-time is undisputed: on the negative side, implementations that do not comply with the policy are often vulnerable to cache attacks [Albrecht and Paterson 2016; AlFardan and Paterson 2013; Ronen et al. 2018]; on the positive side, constant-time implementations are much harder to break (without using advanced micro-architectural features and in particular speculative execution). Furthermore, this practical effectiveness is supported by theoretical justifications: Barthe *et al.* [Barthe et al. 2014] show that constant-time implementations are secure against powerful adaptive adversaries with control over the cache and the scheduler in an idealized model of virtualization.

Cryptographic constant-time is an appealing property to study in the context of secure compilation. On the theoretical side, cryptographic constant-time is an instance of observational non-interference, an information-flow property that reasons about instrumented semantics of programs—see [Barthe et al. 2018; Ngo et al. 2017] for notions of observational non-interference. Therefore, techniques for proving preservation of cryptographic constant-time could form a good starting point for proving preservation of observational non-interference policies and more generally relational properties. On the practical side, building correct, efficient and constant-time cryptographic libraries is very challenging. One problem is that, despite its apparent conceptual simplicity, programming following the cryptographic constant-time policy is difficult, and expert developers occasionally fail to adhere to the policy. Recently, several verification tools have been built and used for checking that popular libraries achieve cryptographic constant-time, see e.g., [Almeida et al. 2016; Rodrigues et al. 2016]. These tools target low-level implementations, both for efficiency reasons and because of the possibility that compilers may break cryptographic constant-time. Unfortunately, checking low-level implementations is not ideal, because it makes the results of the analysis difficult to understand for programmers. In addition, verifying low-level implementations may be less precise, as pointed out in [Blazy et al. 2019], and it may also require more complex analyses, including precise alias analyses.

Pleasingly, many compiler optimizations do preserve constant-time security. In particular, it has been observed recently that both the Jasmin [Almeida et al. 2017] and Low* compilers [Protzenko et al. 2017] preserve constant-time security. Moreover, [Barthe et al. 2018] introduces a general notion of CT-simulation, which combines the usual notion of simulation from compiler correctness and the well-known unwinding lemmas (more specifically, the step-consistent unwinding lemma) from the information-flow literature. This notion can be used to prove preservation of constant-time for a broad range of optimizations, and has been used to formally verify preservation of constant-time for a core language and compiler. This work raises the general question whether a realistic compiler like CompCert preserves cryptographic constant-time.

All results presented in this paper have been mechanically verified using the Coq proof assistant. The complete Coq development is available as a supplementary material. The main contribution of this paper is a machine-checked proof that a mildly modified version of the CompCert compiler preserves cryptographic constant-time. To our best knowledge, our proof is the first of this kind, making CompCert the first formally verified secure compiler for a realistic programming language. The proof rests on several technical contributions:

- we identify the passes of the CompCert compiler that do not preserve constant-time, and adjust them accordingly (see Section 6.2 for a discussion). We also modify CompCert to generate conditional moves, one of the main ingredients used by cryptographers to ensure that implementations are constant-time;

- we formalize the notions of cryptographic constant-time for CompCert intermediate representations, and show equivalence between different notions.
- we develop new proof techniques for proving preservation of cryptographic constant-time. Our proof techniques are designed to minimize proof effort and maximize compatibility with pre-existing proofs for proving compiler correctness. Concretely, we propose a range of methods that make different trade-offs between generality and proof effort, and use different methods appropriate for each pass. In particular, the method from [Barthe et al. 2018] is at the end of our spectrum (most general, most difficult to use) and we use it sparsely;
- we prove that all passes in (modified) CompCert preserve cryptographic constant-time;
- we evaluate the performance of our compiler on representative examples.

This paper is organized as follows. First, Section 2 recalls the required background on the CompCert compiler. Then, Section 3 defines our cryptographic constant-time policy for CompCert and two of its variants. Section 4 explains why CompCert does not preserve our cryptographic constant-time policy and how we modified CompCert so that it preserves it. Section 5 details our new proof techniques and how we applied them to prove that CompCert preserves cryptographic constant-time. Section 6 describes the experimental evaluation of our modified compiler. Related work is discussed in Section 7, followed by concluding remarks.

2 BACKGROUND ON THE COMPCERT VERIFIED C COMPILER

CompCert [Leroy 2009a] is the first commercially available optimizing compiler that is formally verified. The CompCert compiler comes with a machine-checked correctness theorem, conducted using the Coq proof assistant [Inria 2019]; it states that the compilation does not introduce bugs in compiled programs. This section introduces the ingredients of the correctness proof of CompCert passes; they are further detailed in [Leroy 2009b]. First, the section describes the generic semantic framework used to define the formal semantics of the 10 languages of the compiler. Then, it details the proof techniques underlying the correctness lemmas. Last, it introduces memory injections that formalize how memory is transformed by compiler passes.

2.1 Transition Semantics

In CompCert, semantics are defined using an operational style. For high-level languages, the formal semantics combines a big-step semantics for expressions and a small-step semantics for statements. Low-level languages are defined by small-step semantics. An operational semantics defines an execution relation between semantic states and associates to each program the set of all its possible behaviors. Three kinds of behaviors observe terminating, diverging and going-wrong programs. The trace of input/output external actions (i.e., calls to library functions, loads and stores on global `volatile` variables) performed by a program is also observed during program execution. A program is *safe* when its execution either terminates or diverges. Traces and behaviors are defined on top of Figure 1, where ε denotes an empty trace, and ν denotes a single external input/output event. A trace is a list of events; this list may be finite or infinite.

Small-step semantics of the CompCert languages are specified using a generic notion of a one-step transition relation $\rightarrow$ between a global environment G which does not change during execution, an initial program state S, a final program state S' and a trace t of input/output actions [Leroy 2009b]. Given an environment G, the transition from S to S' generates external events that are recorded in trace t. The definitions of the global environment and of the state differ from one language to another. They at least involve a local environment and a memory state M (shared by all the languages of the compiler). For example, in the `RTL` language [1], the local environment is a

[1] Figure 11 recalls the languages and passes of CompCert.

Behaviors

$t ::=$	$\varepsilon \mid v$.t		finite traces (inductive)
$T ::=$	$\varepsilon \mid v$.T		finite or infinite traces (coinductive)
$B ::=$	`terminates`(t, n)		termination with trace t and exit code n
$\mid$	`diverges`(t)		divergence with finite trace t
$\mid$	`reacts`(T)		divergence with trace T
$\mid$	`goes_wrong`(t)		blocking execution with trace t

Semantic judgements

$G \vdash S \xrightarrow{t} S'$	(step transition)
$G \vdash S \xrightarrow{t}^{*} S'$	(star transition)
$G \vdash S \xrightarrow{t}^{+} S'$	(plus transition)
$G \vdash S \xrightarrow{t}^{n} S'$	(counted transitions)
$G \vdash S \xrightarrow{T} \infty$	(infinite transitions)

Fig. 1. Transition semantics defined in CompCert (from a generic step transition)

stack frame Σ (representing pending function calls) and the program states are either regular states $\mathcal{S}(\Sigma, f, \sigma, pc, R, M)$ carrying the currently-executing function f with stack pointer σ, program counter pc, map R from registers to values, or call states $C(\Sigma, Fd, \vec{v}, M)$ reached when calling a function, or return states $\mathcal{R}(\Sigma, v, M)$ reached when returning from a function.

CompCert defines several transition relations from the generic step relation, together with their properties: the star (resp. plus) relation denotes its reflexive transitive closure (resp. transitive closure); the$\rightarrow^{n}$ relation counts finite numbers of transitions, and infinite transitions are represented by the relation$\rightarrow \infty$ of Figure 1. A diverging program execution observes infinite or finite traces (e.g., an infinite loop that does not observe external input/output events). Moreover, a generic notion of transition semantics is defined in CompCert. In addition to the type of program states, the type of global environments and the transition relation, the semantics defines two predicates representing the initial states of a program and the final states of a terminating program execution. The first predicate checks that the memory and global environment are initialized, and that there is a `main` function in P with a correct signature. In the remainder of this paper, for the sake of brevity, we omit the global environment G in the transition relations.

2.2 Simulation Relations Between Program Executions

In `C` and in CompCertC, the source language of CompCert, evaluation orders are not fully specified (meaning that the compiler is allowed to choose one of the possible evaluation orders). An example is an expression made of calls to several functions: the expression is safe and the calls can be evaluated in any order that the compiler may choose. CompCertC is the only non-deterministic language of CompCert in that respect. For this reason, the correctness theorem of CompCert is stated as follows: if the compiler produces code C from source S, without reporting a compile-time error, then every observable behavior of `C` is a possible behavior of S. The standard proof technique to conduct this proof is a backward simulation (i.e., every behavior of C is also a behavior of S).

It is hard to build backward simulation relations in the frequent case where a step in the original program is implemented by several steps in the compiled program. For passes that preserve the amount of nondeterminism, it is easier to reason on forward simulations (i.e., stating that every behavior of S is also a behavior of C). A backward simulation from C to S can be constructed from a forward simulation from S to C when S is receptive and C is determinate [Sevcík et al. 2013]. S is

$$\begin{tikzcd}
s_1 \arrow[r, "t"] \arrow[d, dash, dotted, "\approx"'] & s_2 \arrow[d, dash, dotted, "\approx"] \\
\sigma_1 \arrow[r, "t", "+" near end] & \sigma_2
\end{tikzcd}
\qquad
\begin{tikzcd}
s_1 \arrow[r, "t"] \arrow[d, dash, dotted, "\approx"'] & s_2 \arrow[dl, dash, dotted, "\approx"] \\
\sigma_1 &
\end{tikzcd}
\quad \text{with } m(s_2) < m(s_1)$$

Fig. 2. Forward-simulation diagram with measure. Black lines are hypotheses, red lines are conclusions.

receptive when each step produces at most one input/output action in the observed trace, and when two matching traces t_1 and t_2 (i.e., they represent the same actions in the same order) are such that any step $s \xrightarrow{t_1} s_1$ implies that there exists a state s_2 such that we have $s \xrightarrow{t_2} s_2$. C is determinate when each step produces at most one input/output action in the observed trace, and when two matching traces t_1 and t_2 observed from a same initial state represent a same step (i.e., a same final state and a same trace).

So in CompCert, the only backward simulation proof of a compiler pass is the correctness proof of the transformation of CompCertC programs into determinized programs. All other passes are proved correct using a forward simulation. The new proof techniques we define in this paper rely on forward simulation diagrams (i.e., we do not need to update backward simulations of CompCert).

Furthermore, a C undefined behavior means that anything can happen when this behavior is observed, and a C compiler is free to remove computations that cause an undefined behavior (e.g., an assignment including a division by zero that is detected as dead code by an optimization). Thus, in compiler verification, the correctness theorem only applies to safe programs (see Section 2.1). For this reason, our new proof techniques only consider safe programs as well.

Last, the most general forward-simulation diagram is defined in CompCert as follows. The correctness proof of a compiler pass from language L_1 to language L_2 relies on a forward simulation diagram shown in Figure 2 and expressed in the following theorem. Given a program P_1 and its transformed program P_2, each transition step in P_1 with trace t must correspond to transitions in P_2 with the same trace t and preserve as an invariant a relation $\approx$ between states of P_1 and P_2. In order to handle diverging execution steps and rule out the infinite stuttering problem (that may hapen when infinitely many consecutive steps in P_1 are simulated by no step at all in P_2), the theorem uses a measure over the states of language L_1 that strictly decreases in cases where stuttering could occur. It is generically noted $m(\cdot)$ and is specific to each compiler pass. CompCert simulation relations describe only one execution of a program. This is enough to prove the correctness of CompCert, but not adapted to prove our security property, as it involves two different executions of a program. We explain in Section 3 how we circumvent this problem.

2.3 Memory Injections

The memory model of CompCert is shared by the semantics of all the languages. Memory states are collections of blocks, each block being an array of abstract bytes. A block represents a C variable or an invocation of `malloc`. Pointers are represented by pairs `(b,i)` of a block identifier and a byte offset `i` within this block. Pointer arithmetic modifies the offset part of a pointer value, keeping its block identifier part unchanged. Values stored in memory are the disjoint union of 64-bit integers, 64-bit floating-point numbers, pointers, and the special value `vundef` representing the contents of uninitialized memory [Leroy et al. 2014].

The memory layout evolves during compilation of a program. Some memory blocks are added, while others are merged. To relate memories before and after these modifications, the memory model of CompCert comes with a central notion of memory injection that formalizes the effect of merging memory blocks together. Memory injections are used to specify the passes that transform the memory layout. The stereotypical example is the construction of stack frames. For example, in `Cminor` a single block contains all the local variables, stored at different offsets. In contrast, at the upper level, each local variable is allocated in its own block. This mapping from local variable blocks to offsets in the stack block in `Cminor` is captured by a memory injection. A memory injection is characterized by an injection function that optionally associates with each block a new block and an offset within that block.

3 CRYPTOGRAPHIC CONSTANT-TIME POLICIES FOR COMPCERT

This section first defines the notion of cryptographic constant-time policy for CompCert. Second, it presents a restriction on program executions that is adapted to our policy. We then give two alternative definitions that use this restriction and show that they are equivalent to the baseline definitions. These alternative definitions are more convenient for our proofs.

3.1 Main Cryptographic Constant-Time Policy for CompCert

Intuitively, we say that a program P is cryptographic constant-time if two of its executions that differ only on their secret inputs induce equal leakage. In this paper, we note $\varphi(s_i, s'_i)$ the fact that two initial states s_i and s'_i share the same values for public inputs of P, but differ on the values of secret inputs of P. Let us note that φ is used to model what parts of the initial state of P are secret; we do not need this information when compiling and benchmarking program execution times. Marking variables as secret is only required when verifying that a program is constant time, but not when compiling it. Our formalization splits the definition of φ into a generic part and a part that is language dependent, by introducing a generic notion of input world but we omit it in the remainder of the paper.

We model the leakages of P as a list of atomic leakages, where an atomic leakage results from a single step of the program execution [Barthe et al. 2018]. An atomic leakage is either the truth value of a condition, or a pointer representing the address of either a memory access (i.e., a load or a store) or a called function. This leakage model is compliant with our constant-time policy. Moreover, to observe leakages in all languages of CompCert, we instrument all their semantics, and enrich the traces of input/output events defined in CompCert by leakages (i.e., we add leakages as a new kind of event). We thus benefit from all the existing definitions and lemmas about traces, that we directly reuse in our proofs.

More precisely, we distinguish 3 semantics: the CompCert semantics with traces enriched by leakages, represented by the full step relation $\xrightarrow{t}$, and two semantics where some events are filtered out from traces. We call them *leak-only* semantics (represented by the $\xrightarrow{t}_l$ step relation) and *compile-only* semantics (represented by the $\xrightarrow{t}_{Comp}$ step relation). A *leak-only* execution is based on the former relation, while a *compile-only* execution is based on the latter. We omit the arrow subscripts when the context makes clear which is considered. In leak-only semantics, the input/output events of the full semantics are filtered out to produce only leakages. In compile-only semantics, the leakages of the full semantics are filtered out to produce only input/output events. It corresponds hence to the original semantics of CompCert. As explained in the remainder of this paper, in our formalization, we either reuse directly standard simulation diagrams (and semantics) of CompCert or define new simulation diagrams involving one of our instrumented semantics. These proof techniques range from simple proofs (see Section 5.2) to tricky proofs requiring sophisticated

$$a\,,\ a' \text{ initial states of } P \implies a \equiv a' (1) \qquad \begin{matrix} a \text{ final state of } P \\ a \equiv a' \end{matrix} \implies a' \text{ final state of } P \qquad (2)$$

$$\begin{matrix} a \xrightarrow{t} b \\ a' \xrightarrow{t} b' \\ a \equiv a' \end{matrix} \implies b \equiv b' (3) \qquad \begin{matrix} a \xrightarrow{t} b \\ a' \xrightarrow{t'} b' \\ a \equiv a' \end{matrix} \implies |t| = |t'| \qquad (4)$$

Fig. 3. Expected properties of the same-point relation ≡

simulation diagrams and peculiar instrumentations in the semantics (see Section 5.5). Last, we write $|t|$ the size of trace t (i.e., the number of its observed events).

Furthermore, we assume that the languages are deterministic. This assumption is appropriate for our intended application as cryptographic implementations are deterministic. Moreover, it does not seem straightforward what it would mean for a non-deterministic C program to be constant-time. The main cryptographic constant-time policy for CompCert is thus stated as follows.

Definition 3.1 (CTS). Let P be a safe program (see Section 2.1). We say that P is constant-time secure w.r.t. φ (abbreviated as *CTS*) if for two initial states s_i, s_i' of P such that $\varphi(s_i, s_i')$ holds, then both leak-only executions starting from s_i and s_i' observe the same leakage (i.e., both leak-only executions terminate with the same trace, or they both diverge with the same trace).

3.2 Other Characterizations of Constant-Time Security

The *CTS* property ensures that differing executions of a safe program have the same control flow. Indeed, conditions of branches are leaked and *CTS* ensures that leaks are the same in these executions. Our proof methodology requires to introduce, for each intermediate language, a notion of a *same-point* relation between states. It expresses that during two executions of a program, two semantic states are related to a same program point. The relation is written ≡ and must satisfy some properties presented in Figure 3. Property 1 states that all initial states of the same program are at the same program point. Furthermore, property 2 states that if a and a' are at the same program point, and one of them is a final state, then the other necessarily is.

The next two properties give some characterizations of the leakage model we consider. Property 3 means that step transitions with equal leakage preserve the same-point relation: if both states a and a' are at a same program point, and both program executions make one step respectively to b and b' while leaking the same leakage (belonging to trace t), then b and b' are at a same program point. It ensures that the leakage reveals enough information to characterize the control-flow of the execution. Property 4 means that if both states a and a' are at a same program point, then both steps from a and a' leak the same amount of information (hence the same size for both traces). This prevents from building an attacker model in which the mere absence of a leakage reveals some information. We do not require the traces to be equal because t and t' may not depend only on control-flow position. In the special case where t is empty, this property ensures that t' is empty too. This property is useful in our methodology and easily satisfied by all CompCert semantics. Another remark is that this property subsumes the non-cancelling assumption made by [Barthe et al. 2018], i.e., that the equality of the leakages of two executions implies the pairwise equality of the leakages of each of their steps. This is a direct consequence of our property.

When such same-point relations exist, they are all consequences of the most informative relation (written $\equiv_{\text{strongest}}$): two states are related if and only if they are reachable from two initial states within the same number of steps and producing the same trace. Although the relation $\equiv_{\text{strongest}}$

is convenient to prove general properties of $\equiv$-related states, it is generally helpful to define a same-point relation for each language in terms of the control-flow components of the execution states. We illustrate that the $\equiv$ relation is consequence of the $\equiv_{\text{strongest}}$ one in the following lemma.

Lemma 3.2. *For all $\equiv$ relations, we have $\equiv_{\text{strongest}} \subseteq \equiv$.*

Proof. Let s and s' such that $s \equiv_{\text{strongest}} s'$. By definition, there exist initial states s_i and s'_i, number of steps n and trace t such that $s_i \xrightarrow{t}{}^{n} s$ and $s'_i \xrightarrow{t}{}^{n} s'$. We show by induction on n that $s \equiv s'$. If $n = 0$, then $s = s_i$ and $s' = s'_i$, and therefore $s \equiv s'$ by property 1.

Otherwise, there exist states s_{n-1} and s'_{n-1}, traces t_1, t_2, t'_1, t'_2 such that $s_i \xrightarrow{t_1}{}^{n-1} s_{n-1} \xrightarrow{t_2} s$, $s'_i \xrightarrow{t'_1}{}^{n-1} s'_{n-1} \xrightarrow{t'_2} s'$ and $t = t_1 \mathbin{+\!\!+} t_2 = t'_1 \mathbin{+\!\!+} t'_2$. By induction hypothesis we have that $s_{n-1} \equiv s'_{n-1}$. By Property 4, we have that $|t_2| = |t'_2|$. We thus have necessarily that $t_1 = t'_1$ and $t_2 = t'_2$. Hence, $s \equiv s'$ by Property 3. □

Given a same-point relation, we characterize in two supplementary ways constant-time security, and show that our three definitions are pairwise equivalent. Our two following definitions are more restricted than *CTS*, but they facilitate our correctness proofs.

Definition 3.3 (CTS^n). Let P be a safe program. P is CTS^n w.r.t. φ if the following holds. Let s_i and s'_i be initial states of P such that $\varphi(s_i, s'_i)$ holds. If the leak-only execution of P from s_i (resp. s'_i) reaches state s (resp. s') in a same number of steps with a leakage trace t (resp. t'), then both traces (hence both leaks) are the same. More formally, for all n, s, s', t, t' such that $s_i \xrightarrow{t}{}^{n} s$ and $s'_i \xrightarrow{t'}{}^{n} s'$, then $t = t'$.

Lemma 3.4. *CTS^n implies CTS.*

Proof. We show that if a program P satisfies CTS^n, then it also satisfies *CTS*. This proof is similar to the standard simulation proofs of CompCert. Let's first assume that the execution of P from s_i terminates with leakage t in n steps, i.e., $s_i \xrightarrow{t}{}^{n} s_f$ and s_f is a final state. We can assume without loss of generality that s'_i has a longer execution, whether terminating or infinite.

We can split this execution in two parts, so that the first part takes also n steps and matches the execution starting from s_i, i.e., $s'_i \xrightarrow{t'}{}^{n} s'_n$. By using CTS^n, we have that $t = t'$. Therefore, $s_f \equiv s'_n$ by Lemma 3.2.

As s_f is a final state, s'_n is also necessarily a final state by Property 2. The execution starting from s'_i is thus also terminating with leakage t' such that $t = t'$.

If the execution of P from s_i diverges with leakage t, then in a similar way, the execution of P from s'_i must diverge with some leakage t' (otherwise, the first execution would have been terminating). By using CTS^n, we have that all prefixes of same size of t and t' are equal, thus $t = t'$. □

Our third definition (called $CTS_{|l|}$) requires leakages to have the same length, without requiring executions to have the same numbers of steps.

Definition 3.5 ($CTS_{|l|}$). Let P be a safe program. P is $CTS_{|l|}$ w.r.t. φ if the following holds. Let s_i, s'_i be two initial states of P such that $\varphi(s_i, s'_i)$ holds. If for all s, s', t, t' such that $s_i \xrightarrow{t}{}^{*} s$, $s'_i \xrightarrow{t'}{}^{*} s'$ and $|t| = |t'|$, then $t = t'$.

Lemma 3.6. *CTS^n and $CTS_{|l|}$ are equivalent.*

Proof. We first prove that CTS^n implies $CTS_{|l|}$. Let s_i and s'_i be two initial states of P such that $\varphi(s_i, s'_i)$, for all s, s', t, t' such that $s_i \xrightarrow{t}{}^{*} s$, $s'_i \xrightarrow{t'}{}^{*} s'$ and $|t| = |t'|$. Without loss of generality, we can

$$\frac{\Sigma \equiv \Sigma' \qquad \#(M) = \#(M')}{\mathcal{S}(\Sigma, f, \sigma, pc, R, M) \equiv \mathcal{S}(\Sigma', f, \sigma, pc, R', M')} \qquad \frac{\Sigma \equiv \Sigma' \qquad \#(M) = \#(M') \qquad |\vec{v}| = |\vec{v}'|}{\mathcal{C}(\Sigma, Fd, \vec{v}, M) \equiv \mathcal{C}(\Sigma', Fd, \vec{v}', M')}$$

$$\frac{\Sigma \equiv \Sigma' \qquad \#(M) = \#(M')}{\mathcal{R}(\Sigma, v, M) \equiv \mathcal{R}(\Sigma', v', M')}$$

Fig. 4. Definition of the same-point relation for the RTL language

assume that the first execution is longer than (or equal to) the other, we can thus cut it short such that there exists $t = t_1 +\!\!+ t_2$. By CTS^n hypothesis, we get that $t_1 = t'$ and therefore $|t| = |t'| = |t_1|$ and $|t| = |t_1| + |t_2|$. It is thus necessary that $t_2 = \varepsilon$, hence $t = t'$.

Proving that $CTS_{|l|}$ implies CTS^n is slightly more challenging. Indeed, let s_i and s'_i be two initial states such that $\varphi(s_i, s'_i)$. For all n, s, s', t, t' such that $s_i \xrightarrow{t}{}^n s$ and $s'_i \xrightarrow{t'}{}^n s'$, we prove that $t = t'$ by induction on n. For $n = 0$, it is true since $t = t' = \varepsilon$. Otherwise, there exist executions $s_i \xrightarrow{t_1}{}^{n-1} s_{n-1} \xrightarrow{t_2} s$, $s'_i \xrightarrow{t'_1}{}^{n-1} s'_{n-1} \xrightarrow{t'_2} s'$ and $t = t_1 +\!\!+ t_2$ and $t' = t'_1 +\!\!+ t'_2$. By induction hypothesis, we have that $t_1 = t'_1$ and therefore $s_{n-1} \equiv s'_{n-1}$ by Lemma 3.2. By Property 4 of the same-point relation, t_2 and t'_2 have same length, and thus t and t' also have same length. Finally, by definition of $CTS_{|l|}$, $t = t'$. □

Lemma 3.7. *CTS implies $CTS_{|l|}$.*

Proof. Let s_i and s'_i be two initial states such that $\varphi(s_i, s'_i)$, for all s, s', t, t' such that $s_i \xrightarrow{t}{}^* s$, $s'_i \xrightarrow{t'}{}^* s'$ and $|t| = |t'|$. t and t' are prefixes (by determinism) of the full traces obtained by considering the full executions from s_i and s'_i which are equal by hypothesis, thus t and t' are equal. □

Theorem 3.8. *The three definitions CTS^n, CTS, and $CTS_{|l|}$ are equivalent.*

Proof. This is a consequence of the three previous lemmas. □

Given these multiple characterizations of constant-time security, we consequently obtain three ways of proving that security is preserved as we show in the following sections where we assume that all considered languages satisfy the same-point relation. Let us note that this relation is only a proof tool. We use it as an invariant of the program execution, which leads to simpler proofs. The relation is not used to formulate the variants of constant-time security, but only to prove them equivalent. We could indeed only use the strongest relation to prove the equivalence. However, it is easier in some cases (e.g., proving same-point congruence defined in definition 5.4) to work with other same-point relations that relate the control flow of states (e.g., the call stacks have a similar shape).

3.3 Defining the Same-point Relation

For each language of CompCert, we defined a same-point relation and proved that it satisfies the properties of Figure 3. These definitions depend on how much information we need to prove; they all relate the control flow of states, but in some cases, we also need to relate the memory layout.

For instance, Figure 4 defines the same-point relation for the RTL intermediate language, where RTL states are defined in Section 2.1. Only states of a same kind can be at a same point. Two program states are at the same point if their stack frames Σ and Σ' match (we omit the definition of $\Sigma \equiv \Sigma'$ for simplicity) and their memories M and M' have the same layout (written as $\#(M) = \#(M')$). More

$$\begin{tikzcd} S \arrow[r, "C"] \arrow[dr, "f"'] & S/_{\equiv} \arrow[d, "g"] \\ & T \end{tikzcd}$$

Fig. 5. Commutative diagram proving that a function f only depends on the control-flow part of the state.

precisely, M and M' have the same number of allocated blocks and the same identifier for the next block to allocate. This definition ensures that states at the same point have a similar memory layout but says nothing about their contents. For regular states, we further require that the states are within the same function f. For called states, we further require the same function definition *Fd* to be called with the same number of arguments.

Defining the same-point relation for assembly programs is slightly more challenging. Indeed, indirect jumps and calls transfer control to an arbitrary computed position. Justifying that two executions follow the same control flow thus requires to prove that the computed return addresses agree. This is particularly difficult as assembly programs *may* tamper with their call stacks and modify return addresses. However, as we are only concerned with assembly programs produced by the compiler, we can prove that calls and returns are properly bracketed.

To this end, we would like to strengthen the semantics of assembly programs to make the call-stack explicit and dynamically check that the targets of return instructions is consistent with this stack. However, assembly being the target language of the CompCert compiler, it is part of its trusted computing base (TCB) and should not be changed lightly. Therefore, we introduce an intermediate *instrumented* semantics with a ghost call-stack and assertions about its consistency. We then prove a simulation which justifies that the instrumentation can be removed (preserving traces, hence behaviors as well as the constant-time property). This implies that this instrumentation is only a proof intermediate and does not belong to the TCB. Finally, we prove the correctness of the compilation from the last intermediate language `Mach` to assembly using the instrumented semantics, embedding in the behaviors of assembly programs the knowledge that they come from the usual compilation process.

3.4 Benefiting from the Same-point Relation: Explicit Control-flow State

Generally, the same-point relation is an equivalence relation. One way to define such an equivalence relation is to define the corresponding quotient: a function C that extracts from a state its control state (usually the program point, the stack of return points, etc.). Many of the proof obligations in this work are of the following form: prove that some function $f : S \longrightarrow T$ is compatible with the same-point relation. In other words, for any states s and s' such that $s \equiv s'$, show that $f(s) = f(s')$. If we can instead define that function from the control state instead of the full state (i.e., to explicitly factor it as $f = g \circ C$, as in the diagram of Figure 5), then all these proof obligations vanish. The compatibility property follows from the type of the function. It is no longer necessary to prove it for each such function. Moreover, it is not even needed to state these properties, as they are mere instances of the congruence of equality.

As an additional benefit, working explicitly modulo same-point avoids relational reasoning: it is no longer necessary to consider two steps from equivalent states, nor two pairs of related states that are pairwise equivalent, etc.

4 A COMPCERT COMPILER THAT PRESERVES CRYPTOGRAPHIC CONSTANT-TIME

Compilers may break the cryptographic constant-time property of a source program. Well-known examples of introduction of `if` statements during compilation are string equality, lazy operations and optimizations on multiplications [Barthe et al. 2018; Kaufmann et al. 2016]. The cryptographic constant-time property may also be broken due to the introduction of memory accesses. For example, when there are not enough available registers, a register allocation pass generates new memory accesses that can induce leakages. This section explains why the CompCert compiler (version 3.4) does not preserve the constant-time policy, and how we modified CompCert 3.4 in order to preserve it. Our modified CompCert targets the `x86` architecture. First, the section details how the instruction selection pass introduces conditional branches in compiled code. Then, the section explains how we modified CompCert so that the compilation does not introduce conditional branches anymore[2]. It relies on the use of a selection operation that is architecture dependent and shared by the languages of the backend of the compiler; it is integrated in CompCert version 3.6.

4.1 Non-preservation of our Policy by the Instruction Selection Pass

Instruction selection is the first pass of the backend compiler. It optimizes `Cminor` expressions to take advantage of combined expressions and addressing modes of the target processor and of processor's capabilities [Leroy 2009b]. `Cminor` expressions are optimized into `CminorSel` expressions, that use machine-specific operators and addressing modes. These optimizations are defined using rewrite rules for each operator. The `Cminor` language supports all logical and arithmetic operators of C. Moreover, conversions between floats and integers are made explicit, using `Cminor` conversion operators. In `Cminor`, there are 5 kinds of constants, 33 unary operators and 40 binary operators. All the following intermediate languages, from `CminorSel` to `Mach`, share the same definition of operators, which are architecture dependent. To handle in a uniform way this large variety of shared operators, a generic notion of backend operator (`Eop op e`) applied to an operand list `e` is defined in CompCert, and reused in the definition of these languages. It roughly represents what the processor can compute in one instruction as well as the addressing modes of the processor. Some examples of these operations are `Eop add [v1;v2]` (addition between two integers), `Eop (intconst n) []` (integer constant) and `Eop (shrximm n) [v]` (signed efficient division by 2^n using a logical shift).

Among the `Cminor` operations are int-to-float and float-to-int conversions. Two of them may introduce conditional branches that are not present in the source program, to work around a limitation of the target instruction set architecture: in `x86`, there is no instruction allowing to convert an unsigned 32-bit integer to a float. The workaround is to emulate the generic source operations with several target instructions, sometimes involving new conditional branches. During instruction selection, there are two conversion operations and several comparison operations that generate conditional branches. Figure 6a shows in `C` syntax how the `unsigned int` to `float` conversion is defined in CompCert. It uses the `signed int` to `float` conversion (represented by the function `floatofints`), and introduces a new conditional branch to handle the conversion of an out-of-bound (i.e., greater than or equal to 2^{31}) unsigned integer value. Indeed, the `signed int` to `float` operator can only be applied on an integer whose value is in the interval $[-2^{31}\ ;\ 2^{31})$. In a similar way, the conversion from a `float` to an `unsigned int` introduces a conditional branch.

Moreover, on 32-bit architectures, comparing 64-bit integers introduces conditional branches. Indeed, in the CompCert memory model, a 64-bit integer is represented by a pair of two 32-bit integers. For efficiency reasons of the generated code, comparing two 64-bit integers consists in first comparing their upper-32 bits, and second in comparing their lower-32 bits only if necessary. Therefore, any comparison between 64-bit integers may introduce new conditional branches, as shown

[2] Then the only conditional branches that appear in the produced assembly program correspond to conditional branches at source level.

```
float floatofintu(unsigned int x) {
  float y = 0x1p31; // 2^31
  if (x < 0x80000000) // 2^31
    return floatofints(x);
  else
    return y + floatofints(x − 0x80000000);
}
```

(a) The unsigned int to float conversion.

```
(xh, xl) = x; (yh, yl) = y;
if (xh $==_s$ yh)
  if (xl $<=_u$ yl) goto lthen;
  else goto lelse;
else if (xh $<_s$ yh) goto lthen;
   else goto lelse;
```

(b) Comparison ≤ for the signed 64-bit integers x and y, using signed (s) and unsigned (u) operators.

Fig. 6. Examples (in C-like syntax) of conditional branches introduced by CompCert.

in the example of Figure 6b. Given two signed 64-bit integers called x and y, it shows the two conditional branches generated when compiling the instruction if (x <= y) goto lthen; else goto lelse;. The syntax (xh, xl) = x; introduces the pair of 32-bit integers representing x.

To work around this issue, we changed the implementations of these operations, using a conditional move (i.e., a branchless choice between two values) instead of conditional branches. Indeed, conditional moves are a standard technique to achieve constant-time security. In CompCert, a conditional move instruction (cmov c rd r) is defined but only at assembly level for x86. It performs a move from one integer register r to another register rd, only if condition c holds; otherwise, the move is not performed. This cmov instruction already exists in CompCert, but a user may not directly use it, as it is only generated during the compilation of the shrximm operation.

4.2 Modification of the Instruction Selection and Generation of Assembly Passes

In order to remove the introduced conditional branches, we define a new architecture-dependent selection operation, shared by the intermediate languages of the backend compiler. This section first details this operation and its compilation to the cmov instruction. Then, it explains the further changes we performed on the instruction selection pass, so that it only generates branchless instructions.

4.2.1 A New Selection Operation. Eop (select c ty) [e1;e2] returns the value of type ty of the expression (if c then e1 else e2) when both expressions e1 and e2 are of type ty, and the undefined value vundef otherwise. However, unlike a conditional expression, this operation has a strict semantics: both e1 and e2 are always evaluated, whatever the value of the condition is. Note that the condition c does not belong to the argument list, mainly because of the peculiar way conditions are evaluated on x86 assembly (i.e., using flags registers). The type annotation ty allows the selection operator to be typed like any other arithmetic operator. These typing passes are performed by CompCert at RTL and LTL levels (where most optimizations are performed) to guarantee that operations (Eop op e) are applied to the right number of arguments and that the arguments are of the correct type. Thanks to the ty annotation, we benefit from these typing passes and did not need to change or to add special typing rules to ensure that Eop (select c ty) is typed in a safe way.

The new selection operation is compiled down to the cmov instruction on x86 architecture. We chose to add our selection operation to the generic backend operations because it induces minimal changes in CompCert and does not require modifying directly the syntax of the languages of the backend. In a similar way, thanks to smart constructors, the changes in the compiler passes were minimal and so were the associated correctness proofs.

The selection operation is compiled to the cmov instruction, during the last compiler pass, from Mach to assembly language. At Mach level, res <− Eop (select c ty) [r1;r2] operates over two registers r1 and r2, and stores the result in a register res. In order to generate efficient code and to apply

```
float floatofintu(unsigned int x) {
  C = 0x80000000; // 2^31
  a = select(x < C, 0, −C);
  b = select(x < C, x, x − C);
  return floatofints(b) − floatofints(a); }
```

(a) unsigned int to float conversion.

```
(xh, xl) = x; (yh, yl) = y;
if (select(xh ==_s yh, xl <=_u yl, xh <_s yh))
  goto lthen;
else
  goto lelse;
```

(b) Comparison x ≤ y

Fig. 7. Constant-time implementations (in C-like syntax) of the examples of Figure 6.

CompCert optimizations (e.g., register allocation) on the selection operation as well, select is a two-address operation (i.e., its first argument r1 and its result res lie in the same location). For an integer selection operation (when ty = int), this is done directly using one cmov instruction and the negation of the condition c. So, the Mach operation r1 <– Eop (select c ty) [r1;r2] is compiled down to the x86 instruction cmov (negate c) r1 r2, and the compilation from the CminorSel operation Eop (select c ty) [e1;e2] to the Mach operation Eop (select c ty) [r1;r2] does not require further changes in CompCert.

Moreover, the syntax of the cmov instruction requires c to be a testable condition (usable for a conditional jump), which in our case requires to transform c because of the way comparisons proceed on x86 architecture. More precisely, the x86 hardware provides a way to handle most comparisons of the source language; we call them testable conditions. Two comparisons (namely equality and inequality between floating point numbers) cannot be handled by the hardware, and require a software workaround. The idea is to split these comparisons into several testable conditions, and to combine them with logical and or or operators. Therefore, as the case of equality between floating point numbers introduces a logical and in the condition, we then need to compile the operation r1 <– Eop (select (c1 && c2) ty) [r1;r2]. It is done with the following sequence: cmov (negate c1) r1 r2 ; cmov (negate c2) r1 r2. Finally, the case of inequality (≠) between floating point numbers introduces a logical or in the condition. We chose to negate the condition (hence to introduce a logical and), which we then compile in a similar way as the previous case) and swap the operands of the selection operation.

4.2.2 *Taking Floats into Account.*

Section 4.2.1 only describes the compilation of the integer selection operation. Indeed, the cmov instruction only operates over integer registers, and there is no x86 conditional move instruction over floating point registers. This requires more work to replace the conditional branches introduced by the instruction selection pass by branchless instructions. Directly replacing the conditional branch with a selection operation in the unsigned int to float conversion would require a floating point selection operation. Instead of implementing such an operation, we were able to only use the integer selection operation, as shown in Figure 7a. In this example, the conversion from unsigned int to float first computes the values of temporaries a and b using two integer selection operations, and these values are converted to floating point values. The value of a is always in the range of the float to the signed int operation, and so is the value of b, with respect to the condition x < C. Figure 7b shows our implementation of the comparison of 64-bit integers, which uses the new selection operation instead of conditional branches.

5 PROVING THE PRESERVATION OF OUR CONSTANT-TIME POLICY

Cryptographic constant-time preservation is a property about the *leak-only* semantics $\rightarrow_l$ (see Section 3.1) of a source and a target programs. Existing CompCert simulation diagrams deal with the *compile-only* semantics $\rightarrow_{Comp}$. Our proof engineering strategy is to benefit as much as possible from the proof scripts of these diagrams. First, this section details the main theorem of constant-time preservation. Then, it presents the four proof techniques we use to prove this theorem; except the last one, these are relaxations of the classical diagram that we prove on the *full* step relation $\rightarrow$. Then we derive, on one hand the standard compiler correct forward simulation diagram for the *compile-only* relation, and on the other hand a proof of constant-time preservation for the *leak-only* semantics.

5.1 Preservation of Constant-Time Security in CompCert

Our modified version of CompCert preserves our cryptographic constant-time policy, as stated by the following main theorem, which considers a source `Clight` program. `Clight` is a determinized large subset of `C`. As explained in Section 3, we only consider deterministic languages, which `C` and `CompCertC` are not. We thus chose to start our compilation pipeline from the first determinate intermediate language, which is `Clight`. Wary programmers can directly write programs in this subset in order to make sure that the determinization pass from `CompCertC` to `Clight` did not modify their programs and are covered by our theorem.

Theorem 5.1 (Constant-Time security preservation). *Let S be a safe* `Clight` *source program that is compiled into an* `x86` *assembly program T. If S is constant-time w.r.t. φ, then so is T.*

The theorem is proved by leveraging the proof structure already present in CompCert. We prove that each individual compiler pass preserves constant-time security using four proof techniques presented in the remainder of this section. All such proofs are then composed in order to obtain the final theorem. Each proof technique relies on a simulation diagram that we present in the following subsections: trace-preservation simulation (Section 5.2), leakage-erasing simulation (Section 5.3), leak transforming by memory-injection simulation (Section 5.4) and CT-simulation (Section 5.5). All techniques, except CT-simulation, are subtle variations of a classic simulation diagram. CT-simulation is more involved because it adds an extra dimension of interaction (a *cube* diagram).

We detail in Table 1 which proof technique was used for each pass. We explain in Section 6.2 why there are only 17 passes in the table. Six out of those 17 passes only required minimal modifications, as they preserve traces (hence leakages). Another five passes are proved security preserving by using the leakage-erasing diagram. Further two passes are proved through the leak transforming by memory-injection simulation. Only the pass Linearize relies on a full-fledged CT-simulation. Each of the three remaining passes uses a different technique, which is a generalisation of the leak transforming by memory-injection simulation, and requires to reason on trace transformations. This table shows that thanks to our proof techniques, most transformations were relatively convenient to adapt, and heavily benefit from existing proof scripts.

5.2 Leakage Preservation

The first case we consider is one where leakages are preserved by compilation. This case is actually the one covered by compiler correctness and can thus leverage directly the many already existing lemmas and theorems of CompCert. It is directly stated and proved on the instrumented semantics $\longrightarrow$.

Theorem 5.2 (Constant-time security preservation by leakage preservation). *Let S be a safe source program and T be the target compiled program. If S is constant-time w.r.t. φ then T is constant-time w.r.t. φ, provided that the relation $\approx$ between program states is a trace preserving simulation (w.r.t. $\longrightarrow$) as defined in Figure 2.*

Proof. First we observe that such simulation implies that the *leak-only* semantics satisfies a leak-preserving simulation. This simulation is the classic forward simulation used in CompCert

Table 1. Compilation passes and how we prove them constant-time preserving

Compiler pass	Diagram used	Explanation on the pass
Cshmgen	Trace preservation	Type elaboration, simplification of control
Cminorgen	Memory injection	Stack allocation
Selection	Leakage erasing	Recognition of operators and addr. modes
RTLgen	Trace preservation	Generation of CFG and 3-address code
Tailcall	Trace preservation	Tailcall recognition
Inlining	Trace transformation	Function inlining
Renumber	Trace preservation	Renumbering CFG nodes
ConstProp	Trace transformation	Constant propagation
CSE	Leakage erasing	Common subexpression elimination
Deadcode	Leakage erasing	Redundancy elimination
Allocation	Leakage erasing	Register allocation
Tunneling	Leakage erasing	Branch tunneling
Linearize	CT-simulation	Linearization of CFG
CleanupLabels	Trace preservation	Removal of unreferenced labels
Debugvar	Trace preservation	Synthesis of debugging information
Stacking	Memory injection	Laying out stack frames
Asmgen	Trace transformation	Emission of assembly code

and we know it implies behavior preservation. Hence source and target programs share the same leakages. As a consequence, *CTS* at source level implies *CTS* at target level. □

While this simulation is quite basic and restrictive, it is still satisfied by a few transformation passes (6 among 17, see Table 1), including passes that do not optimize conditional branches, nor memory accesses. These passes are then especially easy to adapt to constant-time preservation since we can reuse the same proof script as for the original simulation proof. This is a great time saver for a pass like `RTLgen` (which transforms a structured `Cminor` program into a `RTL` control-flow graph) because its soundness proof is quite verbose. However, many optimizations do not preserve leakages. Indeed, optimizations such as common subexpression elimination (CSE) or constant propagation can remove memory accesses or even branchings if they manage to conclude that a condition always evaluates to the same truth value.

5.3 Leakage Erasing

Some optimizations erase leakages. They are still constant-time preserving as long as their decision to erase this information does not depend on secret values. For instance, in the CSE pass, some memory loads are replaced by reads from local registers: the corresponding memory-access leak is thus erased; but this does not threaten the security of the target program. This intuition is formally defined by the following *leakage-erasing simulation*, illustrated in Figure 8, where the relation between states $\approx_n$ is indexed by a natural number n that predicts how many steps the target execution takes when a source step is observed. The correctness of this prediction is thus an additional proof obligation in the simulation proof. The next definition is again directly stated on the instrumented semantics $\longrightarrow$ that mixes CompCert and leakage events.

Definition 5.3 (Leakage erasing simulation). Let $\approx_n$ be an indexed relation between source and target states, and m be a measure on source states (see Section 2.2). We say that $\approx_n$ is a leakage-erasing simulation w.r.t. m when the three following conditions hold.

$$\begin{tikzcd}
s_1 \arrow[r, "t"] \arrow[d, dotted, dash, "\approx_n"'] & s_2 \arrow[d, dotted, dash, "\approx_{n'}"] \\
\sigma_1 \arrow[r, "\tau", "n" near end] & \sigma_2
\end{tikzcd}$$

$$\tau = t \lor (\tau = \varepsilon \land t \text{ is leak-only})$$
$$n = 0 \Longrightarrow m(s_1) > m(s_2)$$

Fig. 8. Leakage-erasing simulation. Assumptions are in black and conclusions in red.

(1) For every source step $s_1 \xrightarrow{t} s_2$, and every target state σ_1 such that $s_1 \approx_n \sigma_1$, there exists an index n', a target state σ_2 and a target execution $\sigma_1 \xrightarrow{\tau}{}^n \sigma_2$, such that end states are related (i.e., $s_2 \approx_{n'} \sigma_2$), $m(s_1) > m(s_2)$ if there is no target step (i.e., $n = 0$), and either (preservation as in Section 5.2) $\tau = t$, or (erasure) $\tau = \varepsilon$ and t is leak-only.
(2) For every source initial state s_i, there exists an index n_i and a target initial state σ_i such that we have $s_i \approx_{n_i} \sigma_i$.
(3) For all source and target states s and σ such that $s \approx_n \sigma$, we have s is a final source state iff σ is a final target state.

Note that this definition implies that the compile-only semantics satisfies the standard forward simulation (compiler correctness is preserved).

We make sure that the prediction of n does not depend on secret by requiring it will only depend on the control state. This property is called *step counting prediction is same-point congruent* and defined as follows.

Definition 5.4 (Same-point congruence). Given a *same-point* $\equiv$ relation for source and target languages, an indexed relation $\approx_n$ between source and target states is said *same-point congruent* if for all source states s, s', and target states σ, σ' such that $s \equiv s'$, $\sigma \equiv \sigma'$, $s \approx_n \sigma$ and $s' \approx_{n'} \sigma'$, we have $n = n'$.

As previously noted in subsection 3.4, the same-point congruence holds if the index can be defined as a function of the control-flow parts of the source and target states. Furthermore, same-point congruence implies that the initial index for all source initial states is the same, as all initial states are at the same point (it is noted as n_0 in the rest of the paper).

Example 5.5. The instruction selection preserves leaks except when it replaces a function call by a builtin call. Applying our leakage erasing methodology here consists in the following steps.

(1) Adding a provably correct step counter. The simulation relation consists in 5 cases and the step counter (0 or 1) can be easily guessed looking at the current instruction in the source state.
(2) Proving a leakage-erasing simulation. We reuse exactly the same original simulation script, except a few systematic generic tactics for proving the correctness of the step counter, plus the specific case where a leak is erased.

As a conclusion, the notable effort for this pass is focused on proving the correctness of the new selection operation we introduced (see Section 4), not its security.

More advanced counting is sometimes necessary. Several choices made by the compiler may depend on heuristics; the correctness proof of the compiler should not take into account the details of such heuristics. Instead, the compiler should be correct whatever heuristic is used to guide it. Therefore in the correctness proofs, these choices are often *existentially* quantified (i.e., in a non-constructive way). However, to predict the number of target execution steps, it is necessary to discover what optimizations were applied.

Example 5.6. For instance, the `getparam` instruction of the `Mach` language is normally implemented in assembly using two memory loads: one fetches the address of the caller's stack frame into register A and the second fetches into register B the value of the parameter from the stack frame using the contents of register A as base pointer. When the compiler is able to prove (through a heuristic) that register A already holds the correct value, the first load is not emitted. Under such circumstances, given a step corresponding to the execution of a `getparam` instruction, how to predict how many assembly steps will match it and how many memory accesses will occur? In this specific case, it is possible to recover the choice made at compile time by looking at the generated assembly code: if there is no load to register A at the current program point, then it has been optimized away.

The following theorem states that constant-time security is preserved by a leakage-erasing simulation. Its proof is postponed at the end of Section 5.5.

THEOREM 5.7 (CTS PRESERVATION FROM LEAKAGE-ERASING SIMULATION). *Let S be a safe source program and T be the target compiled program. If S is constant-time w.r.t. φ then T is constant-time w.r.t. φ, provided that $\approx_n$ is a leakage-erasing simulation and is same-point congruent.*

Because the definition of leakage-erasing simulation is very close to the standard CompCert trace-preservation diagram, we were able to pleasantly reuse most of the existing proof scripts. For each simulation, the original proof starts with a case study on the current source instruction; if the instruction preserves the trace we reuse the same script, otherwise we prove that it erases all the leakages, which is generally an easy proof task and concerns very few instructions.

On the other hand, it may be quite difficult to write out the prediction for the number of steps taken by the target state when the corresponding source state takes one step. The process is however *proof driven* as this index is already embedded in the proof in CompCert. It is then needed to understand the proof details, which can be quite involved. For instance, the proof script for the `Cminorgen` pass uses dependent inductions within the main induction on the source steps. This indicates what the predicted number of steps depends on. In such a case of a recursive simulation, the index depends on the proof depth.

This simulation is satisfied by many CompCert passes (5 among 17, see Table 1). Still, there remain passes that require more advanced reasoning. We describe them in the upcoming subsections.

5.4 Memory Injections Transform Leakages

Memory injections (introduced in Section 2.3) deserve a specific proof strategy. Passes that rely on them do not preserve leakages because a memory access in a source program may be transformed into a memory access to a different block and offset at the target level. Still, there exists a *leakage transformation* that maps the source leakage trace to the target leakage trace. However, this transformation is not static: it may depend in particular on the current memory injection. We dedicate a specific proof strategy for this situation. For these passes, each simulation relation contains an existentially quantified injection that relates memory addresses between source and target memories. During the execution, injections are updated to take into account new memory allocations.

CompCert provides a generic framework for reasoning on memory injections. We extend it by providing explicit *injection updates* every time CompCert proves a lemma starting with *there exists an injection such that.* These update operators are then used to explicitly build the next injection during the simulation proof. Hence, instead of having a simulation relation $\approx$ whose definition contains an existentially quantified injection ι, we make this injection ι appear in the signature of the simulation relation.

For example, a (lockstep) diagram like $p \longrightarrow p' \land p \approx q \Rightarrow \exists q', q \longrightarrow q' \land p' \approx q'$ becomes the diagram $p \longrightarrow p' \land p \approx_\iota q \Rightarrow \exists q', q \longrightarrow q' \land p' \approx_{F(p,q,\iota)} q'$, where F builds the next memory injection using the generic injection updates we added.

Definition 5.8 (Leak-transforming by memory-injection simulation). Given an indexed relation $\approx_{n,\iota}$ between source and target states, a measure m on source states, and F a mapping that given source and target states s and σ, and a memory injection ι computes a new memory injection $F(s, \sigma, \iota)$. We say that $\approx_{n,\iota}$ is a *leak transforming by memory-injection simulation* w.r.t. m and F when:

(1) for every source step $s_1 \xrightarrow{t} s_2$, and every target state σ_1 such that $s_1 \approx_{n,\iota} \sigma_1$, there exists an index n', a target state σ_2 and target execution $\sigma_1 \xrightarrow{\tau}{}^{n} \sigma_2$, such that end states are related $s_2 \approx_{n',F(s_1,\sigma_1,\iota)} \sigma_2$, traces t and τ are such that they are related by ι if they are both memory access leaks and equal otherwise[3], and $m(s_1) > m(s_2)$ if there is no target step (i.e., $n = 0$);
(2) there exists an index n_0 and a memory-injection ι_0 such that for every initial source state s_i, there exists an initial target state σ_i such that $s_i \approx_{n_0,\iota_0} \sigma_i$;
(3) for all source and target states s and σ such that $s \approx_{n,\iota} \sigma$, we have s is a final source state iff σ is a final target state;
(4) the step counting prediction is same-point congruent: for all source states s_1, s_2 at same point (i.e., $s_1 \equiv s_2$) and target states σ_1, σ_2 at same point (i.e., $\sigma_1 \equiv \sigma_2$), if s_1 and σ_1 (resp. s_2 and σ_2) are in simulation relation $s_1 \approx_{n_1,\iota_1} \sigma_1$ (resp. $s_2 \approx_{n_2,\iota_2} \sigma_2$), then $n_1 = n_2$;
(5) for all source states s_1, s_2 at same point ($s_1 \equiv s_2$ holds) and target states σ_1, σ_2 at same point ($\sigma_1 \equiv \sigma_2$ holds), if s_1 and σ_1 (resp. s_2 and σ_2) are in simulation relation $s_1 \approx_{n_1,\iota} \sigma_1$ (resp $s_2 \approx_{n_2,\iota} \sigma_2$), then $F(s_1, \sigma_1, \iota) = F(s_2, \sigma_2, \iota)$.

Let us note that the definition concerns the full instrumented semantics $\longrightarrow$ but if we filter it on compile-only event, we recover the compiler correctness forward diagram. Let us also note that we do not treat symmetrically step numbers and memory injections in (5.8.(4)) and (5.8.(5)). Property (5.8.(4)) is easier to prove than introducing a *step number transformation*, but such a simplified situation for memory injection is unfortunately not provable in CompCert (due to a lack on information since several memory injections could satisfy a given simulation relation).

The following theorem states that constant-time security is preserved by a leak transforming by memory injection simulation. Its proof is postponed at the end of Section 5.5.

Theorem 5.9 (CTS preservation from leak transforming by memory injection simulation). *Let S be a safe source program and T be the target program obtained by compilation. If S is constant-time w.r.t. φ then T is constant-time w.r.t. φ, provided that $\approx_{n,\iota}$ is a leak transforming by memory injection simulation.*

One of the difficulties in setting up this simulation pass is that when defining the injection updates, we need to be careful that the function only uses things that are preserved by the same-point relation in order to be able to prove 5.8.(5). For instance, it should only use aspects of the memory layout and not the memory state itself as explained in Section 3.3. Furthermore, whenever a lemma about memory injection is used in the original proof script, we replace it with the more explicit injection update version. The rest of the proof becomes relatively straightforward for the two compilation passes that use this simulation.

[3] More specifically, either t and τ are both empty or equal boolean leakages, or they are both function call (respectively, memory access) leakages using pointers p and ρ, and $\iota(p) = \rho$.

This simulation diagram can be generalized to any datatype that is existentially quantified in the original simulation but is an argument to the security preservation proof. To justify the same-point congruence of this argument, we must make explicit how it evolves during the execution and prove that this evolution cannot be influenced by secret data. The proof for the Inlining pass uses this simulation with dynamic trace transformation.

More advanced passes like assembly generation require inserting new leakages in the target trace. This specific pass implements parameter passing (communication from the caller to the callee function) through memory accesses in the caller's stack frame. Nonetheless, these new leakages are statically predictable: they may only depend on the source code, the source leakage, or the public part of the execution states. Indeed, while the execution state may hold sensitive value, the *same-point* relation characterizes a view of the state that cannot be influenced by secret data. To accommodate for such cases, we generalize what we do with memory injection and program a static prediction function F that predicts the target trace from the control-flow part of the execution states and the source trace. The constant-time preservation of this method is ensured by the CT-simulation presented in the next section.

5.5 CT-simulation with a Cube Diagram

Previous methods allow intensive reuse of existing CompCert proof scripts because they slightly strengthen classical simulation diagrams. In last resort, we rely on a full-fledged CT-simulation. For this purpose, we adapt the CT-simulation from [Barthe et al. 2018] under our strengthened notion of same-point relation and its related properties. It is directly phrased using the leak-only semantics $\to_l$, but we omit the subscript. A CT-simulation can be split into two parts, namely a *counting simulation* and a CT-diagram. These are defined as follows. We consider a general notion of simulation relation $\approx_{n,h}$ parameterized by a step-number prediction n and a *history* h. The exact nature of histories is kept abstract here. It can be instantiated with memory injections or more compound structures (for the inlining and assembly generation passes).

Definition 5.10 (Counting simulation). Let $\approx_{n,h}$ be an indexed relation between source and target states, m be a measure on source states, F be a mapping that given source and target states s and σ, and a history h, computes an updated history $F(s, \sigma, h)$. Let F_i be a mapping that given source and target states s and σ computes an initial history $F_i(s, \sigma)$. We say that $\approx_{n,h}$ is a counting simulation w.r.t. m, F and F_i when the four following conditions hold.

(1) For every source step $s_1 \xrightarrow{t} s_2$, and every target state σ_1 such that $s_1 \approx_{n,h} \sigma_1$, there exists an index n', a target state σ_2 and a target execution $\sigma_1 \xrightarrow{\tau}^{n} \sigma_2$, such that end states are related (i.e., $s_2 \approx_{n',F(s_1,\sigma_1,h)} \sigma_2$) and $m(s_1) > m(s_2)$ if there is no target step (i.e., $n = 0$).
(2) There exists an index n_i such that for every initial states s_i and σ_i, we have $s_i \approx_{n_i,F_i(s_i,\sigma_i)} \sigma_i$.
(3) For all source and target states s and σ such that $s \approx_{n,h} \sigma$, we have s is a final source state iff σ is a final target state.
(4) For all source states s_1, s_1' at a same point (i.e., $s_1 \equiv s_1'$) and target states σ_1, σ_1' at a same point ($\sigma_1 \equiv \sigma_1'$), if s_1 and σ_1 (resp. s_1' and σ_1') are in simulation relation i.e., $s_1 \approx_{n_1,h} \sigma_1$ (resp. $s_1' \approx_{n_1',h} \sigma_1'$), then $F(s_1, \sigma_1, h) = F(s_1', \sigma_1', h)$ and $F_i(s_1, \sigma_1) = F_i(s_1', \sigma_1')$.

The counting simulation is very similar to the previous leak transforming by memory-injection simulation, except that source and target leakages do not need to be related anymore. The most difficult part, property 5.10.(1) is generally proved reusing the original CompCert simulation script, with little modifications. The second component, the CT-diagram, is a *cube* simulation, defined as follows and depicted in Figure 9.

$$\begin{tikzcd}
 & s'_1 \arrow[rr, "t"] \arrow[dd, dotted, no head, "\approx_{n'_1,h}"'] & & s'_2 \\
s_1 \arrow[ur, phantom, "\equiv" sloped] \arrow[rr, "t"] \arrow[dd, dotted, no head, "\approx_{n_1,h}"'] & & s_2 & \\
 & \sigma'_1 \arrow[rr, "\tau'", "n'_1"{pos=0.95}] & & \sigma'_2 \\
\sigma_1 \arrow[ur, phantom, "\equiv" sloped] \arrow[rr, "\tau", "n_1"{pos=0.95}] & & \sigma_2 &
\end{tikzcd}$$

with $\tau = \tau'$ and $n_1 = n'_1$

Fig. 9. CT-diagram (with missing conclusion arrows, for clarity purposes). Assumptions are in black.

Definition 5.11 (CT-diagram). Given an indexed relation $\approx_{n,h}$ between source and target states, we say that a CT-diagram is satisfied by the source and target programs when:

- $\approx_{n,h}$ is a counting simulation with respect to some functions m, F and F_i,
- for all $s_1, s'_1, s_2, s'_2, \sigma_1, \sigma'_1, \sigma_2, \sigma'_2, t, \tau, \tau'$ such that:
  - the two following simulation relations hold: $s_1 \approx_{n_1,h} \sigma_1$ and $s'_1 \approx_{n'_1,h} \sigma'_1$,
  - the two following same-point relations hold: $s_1 \equiv s'_1$ and $\sigma_1 \equiv \sigma'_1$,
  - $s_1 \xrightarrow{t} s_2$ and $s'_1 \xrightarrow{t} s'_2$,
  - $\sigma_1 \xrightarrow{\tau}{}^{n_1} \sigma_2$ and $\sigma'_1 \xrightarrow{\tau'}{}^{n'_1} \sigma'_2$,

  we have that $\tau = \tau'$, $n_1 = n'_1$.

This definition does not explicitly require that $s_2 \equiv s'_2$ and $\sigma_2 \equiv \sigma'_2$, because it is a direct consequence of Property 3.

Theorem 5.12 (Constant-time security preservation from CT-diagram). *Let S be a safe source program and T be the target compiled program. If S is constant-time w.r.t. φ then T is constant-time w.r.t. φ, provided that there is a CT-diagram between S and T.*

Proof. We prove that T is constant time using the CTS^n definition. We first generalize the statement to be proved before running an induction. For all trace lengths n and measure ranks M, we then prove by lexicographic induction on (n, M) that for all source states s_1, s_2 reachable from initial states in φ relation after the same number of steps, for all target states α_1, α_2 in simulation relation with s_1, s_2 for a same history h (i.e., $s_1 \approx_{k_1,h} \alpha_1$ and $s_2 \approx_{k_2,h} \alpha_2$), if $m(s_1) = M$, if $s_1 \equiv s_2$ and $\alpha_1 \equiv \alpha_2$, if α_1 reaches γ_1 after n steps and produces a trace T_1 ($\alpha_1 \xrightarrow{T_1}{}^{n} \gamma_1$), if α_2 reaches γ_2 after n steps and produces a trace T_2 ($\alpha_2 \xrightarrow{T_2}{}^{n} \gamma_2$), then $T_1 = T_2$. Moreover $k_1 = k_2$ or all states $s_1, s_2, \alpha_1, \alpha_2$ are final states.

If s_1 is final, so are all other states s_2, α_1, α_2 and $T_1 = T_2 = \varepsilon$. Otherwise there exists source steps $s_1 \xrightarrow{t_1} s'_1$ and $s_2 \xrightarrow{t_2} s'_2$. Because S is constant time, we can prove that $t_1 = t_2$. Thanks to the counting simulation hypothesis, there exists states β_1, β_2 and traces τ_1, τ_2 such that $\alpha_1 \xrightarrow{\tau_1}{}^{k_1} \beta_1$ and $\alpha_2 \xrightarrow{\tau_2}{}^{k_2} \beta_2$. Moreover there exists k'_1 and k'_2 such that $s'_1 \approx_{k'_1, F(s_1,\alpha_1,h)} \beta_1$ and $s'_2 \approx_{k'_2, F(s_2,\alpha_2,h)} \beta_2$ hold. Thanks to the CT-diagram, we have that $\tau_1 = \tau_2$ and $k_1 = k_2$.

It remains to prove that $T_1 = T_2$. Thanks to the same-point properties, we deduce that $\beta_1 \equiv \beta_2$ and $s'_1 \equiv s'_2$. Thanks to the counting simulation, we can prove that $F(s_1, \alpha_1, h) = F(s_2, \alpha_2, h)$.

It remains to compare n and k_1 ($= k_2$).

- If $k_1 = 0$ then $M = m(s_1) > m(s'_1)$ and we can rely on the induction hypothesis on $(n, m(s'_1))$ to conclude.

- If $0 < k_1 \leq n$ then T_1 (resp. T_2) can be decomposed into $T_1 = t_1 +\!\!+ T_1'$ (resp. $T_2 = t_2 +\!\!+ T_2'$) for some new traces T_1' and T_2' such that $\beta_1 \xrightarrow{T_1'}{}^{n-k_1} \gamma_1$ and $\beta_2 \xrightarrow{T_2'}{}^{n-k_2} \gamma_2$. Since $n - k_1 < n$, we can rely on the induction hypothesis on $(n - k_1, m(s_1'))$ to conclude.
- If $n < k_1$ then T_1 and T_2 are both prefixes of trace τ_1. Since $\alpha_1 \equiv \alpha_2$, we have $\alpha_1 \xrightarrow{T_1}{}^{n} \gamma_1$ and $\alpha_2 \xrightarrow{T_2}{}^{n} \gamma_2$; the same-point properties imply that $T_1 = T_2$ (by induction on n).

□

This theorem is similar to the one presented in [Barthe et al. 2018]. They exclusively use the CT-diagram to prove that multiple transformation passes preserve constant-time security for a toy compiler for a core imperative language. We however mainly use it to prove the correctness of the previous methods which allows us to avoid most of the required relational reasoning when proving that compilation passes preserve security. So, we detail the proof of Theorem 5.9 that was postponed in Section 5.4 as it uses Theorem 5.12.

Proof of Theorem 5.9. We show that the existence of a leak transforming by memory injection simulation $\approx_{n,\iota}$ (with respect to some functions m, F) implies the existence of a CT-diagram and conclude by Theorem 5.12. First we build a counting simulation. A history is a memory injection here. Functions m and F are given by the leak transforming by memory injection simulation. Function F_i is defined by $F_i(s, \sigma) = n_0$.

Then we prove the CT-simulation. We assume $s_1 \approx_{n_1,\iota} \sigma_1$, $s_1' \approx_{n_1',\iota} \sigma_1'$, $s_1 \equiv s_1'$, $\sigma_1 \equiv \sigma_1'$, $s_1 \xrightarrow{t} s_2$, $s_1' \xrightarrow{t} s_2'$, $\sigma_1 \xrightarrow{\tau}{}^{n_1} \sigma_2$, $\sigma_1' \xrightarrow{\tau'}{}^{n_1'} \sigma_2'$ and we prove $\tau = \tau'$ and $n_1 = n_1'$. Thanks to property 5.8.(4), we know that $n_1 = n_1'$. For proving $\tau = \tau'$, we rely on property 5.8.(1) (and semantic determinism): τ is non-ambiguously defined by t and ι, and τ' is non-ambiguously defined by t and ι, hence $\tau = \tau'$. □

Proof of Theorem 5.7. We show that the existence of a leak-erasing simulation $\approx_n$ (with respect to some function m, and initial counter n_0) implies the existence of a CT-diagram and conclude by Theorem 5.12. We assume that the simulation relation is same-point congruent (definition 5.4).

First we build a counting simulation. We do not need history modification here, so a constant unit history is fine: $h = \emptyset$. Function m is given by the leak-erasing simulation, and functions F and F_i always return the same constant history.

Then we prove a CT-diagram. The equality $n_1 = n_1'$ is a consequence of the same-point congruence hypotheses. Under the CT-diagram hypothesis, we must also prove the trace equality $\tau = \tau'$ but we know that $(\tau = t \vee \tau = \varepsilon)$ and $(\tau' = t \vee \tau' = \varepsilon)$. The same-point properties also imply that $|\tau| = |\tau'|$, so we can conclude the expected trace equality. □

Proving directly a CT-diagram on a transformation pass requires relational reasoning. For instance we chose to rely on a CT-diagram for the RTL linearization pass where some boolean leakages are negated during the pass. This is explained by the fact that the compiler may decide to put the most likely branch just after the current block. At source level, it corresponds to a simple transformation from `if c then a else b` to `if not c then b else a`. This is the only pass we directly prove with a CT-diagram and the proof experience was not especially pleasant because the diagrams are difficult to exploit (we must in particular exploit two many-step judgements at target level and explain their effects).

6 EXPERIMENTAL EVALUATION AND LIMITATIONS

Section 4 explains how we modified CompCert. The proof effort amounted to a 14% increase in the size of the formal development (around 6k additional lines of spec and 7.5k lines of proof) and

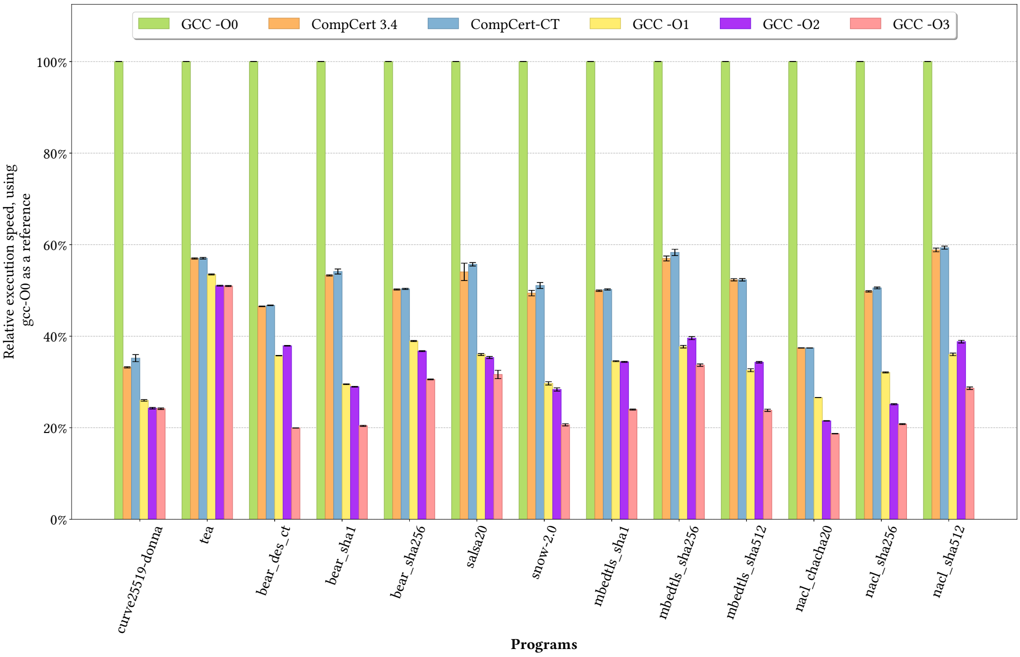

Fig. 10. Relative execution times of our benchmark. We compare the original CompCert, our modified CompCert, and gcc from −O0 to −O3. We normalized the measured execution times with the execution times of gcc −O0. The error bars represent the 99% confidence intervals of our measurements.

took about half a person year. This section first presents the results of experiments evaluating the performance of our constant-time preserving CompCert. Then, it presents the limitations of our work.

6.1 Experimental Evaluation

We carry our experimental evaluation on selected examples from the literature. We note that our experimental evaluation is primarily used to validate that our approach is reasonable. However, a systematic and extensive evaluation of the impact of our compiler, or more generally of the CompCert compiler, on cryptographic libraries (including widely deployed libraries such as [OpenSSL 2019] or repositories such as [SUPERCOP 2019]) is left for future work.

We first compare our version of CompCert to the original CompCert (version 3.4), and to gcc, with and without optimizations. We test these compilers on a benchmark of common cryptographic programs that were shown to be constant-time in [Almeida et al. 2016; Blazy et al. 2019]. They include cryptographic primitives such as an implementation of elliptic curve arithmetic operations over Curve25519 [Bernstein 2006; Langley 2015], and TEA [Wheeler and Needham 1994], together with implementations from commonly used cryptographic libraries such as NaCl [Bernstein et al. 2012] and mbedTLS [ARM 2016]. These are C implementations that we experiment with in order to evaluate our compiler, but it should be reminded that if performance is an issue, it is generally better to use hand-optimized assembly code at the cost of portability.

We first measured the execution times (using an Intel i7-8550U CPU 1.8GHz, with 16GB of RAM), which are shown in Figure 10. We compiled these programs using the original CompCert, our

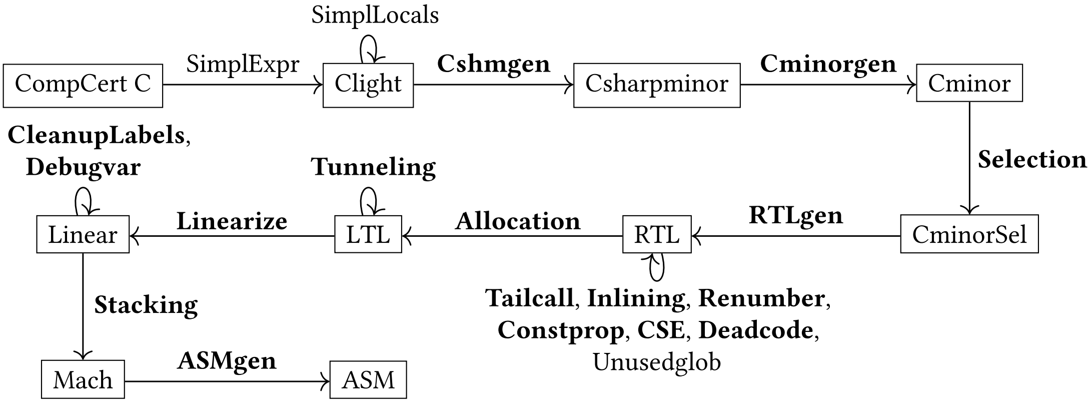

Fig. 11. The CompCert compilation chain. All transformation passes are semantics preserving, bolded ones correspond to formally proved constant-time security preserving passes.

version of CompCert, and gcc at different levels of optimization. As all programs have a very short execution time, we executed them from 10^6 to 10^9 times depending on the program. We obtain shorter execution times than gcc without optimization. This is a promising result, as cautious users who are wary of too aggressive compiler optimizations breaking constant-time security would use it to compile cryptographic implementations. Yet, gcc −O1 and further optimizations remain more efficient. Moreover, we also noticed that our modified CompCert is as efficient as the original CompCert. On average, programs compiled with our modified CompCert are 1.45% slower than programs compiled with the original CompCert.

We further compared two representative cryptographic primitives of our benchmark (namely NaCl Chacha20 stream cipher and Poly1305 authenticator) against heavily optimized implementations using handwritten assembly or AVX instructions (that are not supported by CompCert) from the OpenSSL [OpenSSL 2019] and HACL* [Zinzindohoué et al. 2017] libraries, and implementations written in Jasmin [Almeida et al. 2017, 2019]. For large message sizes, Jasmin is 20 times faster that CompCert when it runs on Chacha 20, and 5 times faster when it runs on Poly1305. These differences are reduced by a half when comparing HACL* and CompCert. These comparisons show that there is still room for progress in CompCert by adding support for extended instruction sets such as AVX for instance.

6.2 Limitations

Among the 20 compiler passes of CompCert shown in Figure 11, only 17 are present in Table 1. The first compiler pass from CompCertC to Clight is a determinization pass; it is removed as explained earlier (see Section 5.1). Moreover, we encountered difficulties while doing the proofs and had to consequently make compromises: two passes were not proved to be security preserving, namely, SimplLocals and Unusedglob. They are still proved semantics preserving as in the original CompCert.

SimplLocals is the second compiler pass. It pulls local scalar variables out of memory. C-like languages such as Clight are split into expressions and statements. Expressions include variable lookups (pointer dereferencing) and are modeled as memory loads under the hood; they must thus produce leakages as instructed by our leakage model. However, expressions may contain multiple nested variable lookups and may thus produce multiple leakages. Modifying CompCert so that the evaluation of expressions may generate leakages is a large endeavour. We therefore chose instead to normalize the source code so that memory loads can only be found at right hand sides of assignments (e.g., the code `f(*x + *y);` becomes `t1 = *x; t2 = *y; f(t1 + t2);`). This standard transformation is commonly used to facilitate analyses (e.g., the Verasco [Jourdan et al. 2015] and VST [Appel 2011] tools use it). Unfortunately, normalizing code before the SimplLocals pass results in significant performance reduction. We hence compromised by performing the normalization after the SimplLocals pass, and prove security preservation starting after the normalization. This compromise does not affect verification tools like Verasco which operates at `Csharpminor` level to prove constant-time security [Blazy et al. 2019]. Furthermore, we removed special support for some external calls that are not critical for cryptographic programs. Additionally, the generation of some builtins that are implemented directly in assembly is also disabled, as they introduce branchings.

The second compilation pass (called Unusedglob) removes unused global variables. This is the only pass that modifies the initial memory layout, and would thus require further proofs about the global environments which are notoriously difficult to reason about in CompCert, as illustrated by the already quite complex proof for this pass. We have proved the pass to be security preserving with a memory injection diagram *assuming* that we can relate the initial states of the source and target programs. As the optimization does not seem critical, we leave proving that the assumption can be satisfied for future work.

We also decided to lighten the instrumentation burden w.r.t. memory stores by copy. We do not support them because they require adding leakage during argument passing. This restriction means that code with functions that return structures or take them as arguments will not compile for instance. It does not concern our cryptographic benchmarks. It is however also acceptable to pass these structures by pointer.

Our last restriction is that we do not support the compilation of switch statements yet. Indeed, they may be either compiled as a sequence of branching instructions or as a jumptable and a branching. Hence, their compilation may introduce multiple leakages where there was only one before in the first case. This transformation is done in the RTLgen pass and should still be security preserving. Unfortunately, we have not yet managed to write out how leakages are transformed in that case in order to fit into one of our diagrams. As switch statements are not critical for cryptographic code, we leave this for future work.

7 RELATED WORK

Jasmin [Almeida et al. 2017, 2019] is a programming framework for efficient and formally verified cryptographic implementations. Jasmin allows programmers to write “assembly in the head” that is predictably compiled to efficient assembly code. The Jasmin compiler is formally verified, but only for semantic correctness. The authors of [Almeida et al. 2017] argue informally that the Jasmin compiler preserves constant-time. However, the proof is not machine-checked.

Low* [Protzenko et al. 2017] is a subset of F* that has been used to program efficient and portable cryptographic implementations. Low* easily compiles to `C`; the translation is made precise by introducing λow*, a formal model of Low*, and by defining compilation to `Clight`. This translation is proved to preserve behavior and side-channel resistance. However, the proofs do not carry to assembly code and are not machine-checked. Furthermore, their compilation scheme is mainly divided in two parts, the first being trace preserving, and thus similar to our method presented in Section 5.2. The second part may change the memory layout and thus the corresponding leakages. However, their compilation pass does not remove leakages and is thus less general than our methodology. Furthermore, their proof involves introducing an intermediate language that is exactly the same as the source one, but where leakages are reinterpreted to correspond to the ones that will appear in the target language. The security preservation is thus split into proving preservation of side-channel resistance for the reinterpretation and simple trace preservation afterwards, whereas our methodology is able to tackle the issue in one pass.

Our work is most closely related to [Barthe et al. 2018], which introduces the notion of CT-simulation and uses the Coq proof assistant for proving preservation of constant-time security for a core imperative language and basic program transformations. Their approach exclusively relies on CT-simulation, whereas we rely on a range of techniques that achieve different trade-offs between proof effort and generality of the technique. In particular, 1) at the end of our spectrum we formalize the CT-simulation method from [Barthe et al. 2018] but adapt it to CompCert proofs; 2) we also design stronger preservation diagrams that are sufficient for some passes and permit major reuse of existing CompCert proof scripts. In our experience, using such a range of techniques is instrumental for reducing proof effort and scaling to realistic compilers.

Our work is also related to independent efforts by Murray *et al.* [Murray et al. 2016; Sison and Murray 2019] to formally verify using the Isabelle proof assistant preservation of value-dependent non-interference for compiling a core concurrent imperative language to a simple RISC architecture. Although their work is cast in a rather different setting and targets a more traditional information-flow policy, the approach is remarkably similar, in that their first paper develops an approach based on "cube-shaped" definition of secure refinement, and their second paper proposes a sufficient "square-shaped" condition which almost halves the proof effort. Whereas our work targets a realistic compiler, their work is focused on a more modest language, and does not have to deal with the legacy of a large pre-existing development.

Other works focus on type-preservation for type systems which enforce secure information flow [Barthe et al. 2006; Chen et al. 2010; Tedesco et al. 2016]. It is worth noting that the two approaches are incomparable, since the type systems are incomplete and therefore preservation of information-flow typing does not entail preservation of non-interference nor the converse for similar reasons. However, the proof techniques used for showing both kinds of results are remarkably close. Besson *et al.* [Besson et al. 2019] present another approach, for specific compilation passes.

Recently, authors have developed compilers that automatically enforce security [Cauligi et al. 2019; Polikarpova et al. 2016; Wu et al. 2018], both for information-flow policies and constant-time security. Interestingly, the correctness of the compiler from [Cauligi et al. 2019] is cast as type-preserving compilation. Unlike our work which targets an end-to-end compiler, these works focus on selected parts of the compilation chain.

Our work is concerned about preservation of constant-time. However, it is also critical to verify that constant-time is enforced properly in the first place. To this end, several authors have developed fully automated methods. In unpublished work, Langley [Langley 2010] developed a variant of Valgrind to verify that implementations are constant-time. A formally verified type-based analysis for constant-time security of CompCert assembly was proposed in [Barthe et al. 2014]. The analysis relied on a simple alias analysis, which limited its applicability. Later, a more powerful analysis based on the Verasco analyzer was implemented for the `Csharpminor` intermediate language of CompCert [Blazy et al. 2019]. The analysis could handle more examples than [Barthe et al. 2014] but was not verified formally. A more powerful type-based static analysis for LLVM programs is proposed in [Rodrigues et al. 2016] and used to verify a large body of cryptographic algorithms. Independently, a sound and (relatively) complete approach based on product programs has been proposed also for LLVM, and also used to analyze large code bases. More recently, a variant of this approach based on relational Hoare logics has been used for proving constant-time security of Jasmin programs [Almeida et al. 2019]. Further work considers constant-time security of floating point operations [Andrysco et al. 2015, 2018] and hardware implementations [von Gleissenthall et al. 2019].

In a different direction, several works have used the CompCert compiler to produce formally verified cryptographic libraries. For instance, Appel *et al.* [Appel 2015; Beringer et al. 2015; Ye et al. 2017] use the Verified Software Toolchain [Appel 2014] to prove functional correctness of cryptographic implementations, and the CompCert compiler to carry these guarantees to assembly code. Similarly, Zinzindohoué *et al.* [Zinzindohoué et al. 2017] use CompCert to generate assembly from HACL* implementations. These works assume that CompCert preserves constant-time; it would thus seem very beneficial to use our modified compiler to prove that the generated assembly code is indeed constant-time.

8 CONCLUSION

We have proved that a modified version of the CompCert compiler preserves cryptographic constant-time. Our formalization relies on carefully crafted methodologies that maximize proof reuse. In the future, it would be interesting to use our modified compiler to generate provably cryptographic constant-time implementations from [Beringer et al. 2015; Ye et al. 2017] and [Zinzindohoué et al. 2017]. Another possibility would be to extend CompCert with support for vectorization instructions in order to reduce the performance gap with handwritten assembly cryptographic implementations. We also intend to extend our proof methodologies to establish preservation of observational non-interference for a larger class of properties that encompasses constant cost — the execution cost, in an idealized model that computes the number of primitive operations, does not depend on secrets. These properties raise two challenges: they allow branching on secrets and therefore require generalizing our methodology, and they are not leakage cancelling in the sense of [Barthe et al. 2018].

ACKNOWLEDGMENTS

This work is supported by DFF project 6108-00363 from The Danish Council for Independent Research for the Natural Sciences (FNU); by the Office of Naval Research under projects N00014-12-1-0914, N00014-15-1-2750 and N00014-19-1-2292; and by a European Research Council (ERC) Consolidator Grant for the project "VESTA", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 772568). The selection operation presented in Section 4.2.1 was discussed with Xavier Leroy who integrated it into CompCert version 3.6.

REFERENCES

Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. 2018. Exploring Robust Property Preservation for Secure Compilation. In *Computer Security Foundations 2019*. http://arxiv.org/abs/1807.04603

Martin R. Albrecht and Kenneth G. Paterson. 2016. Lucky Microseconds: A Timing Attack on Amazon's s2n Implementation of TLS. In *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 (Lecture Notes in Computer Science)*, Marc Fischlin and Jean-Sébastien Coron (Eds.), Vol. 9665. Springer, 622–643. https://doi.org/10.1007/978-3-662-49890-3_24

Nadhem J. AlFardan and Kenneth G. Paterson. 2013. 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*. IEEE Computer Society, 526–540. https://doi.org/10.1109/SP.2013.42

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *25th USENIX Security Symposium, USENIX Security 16*.

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. 2019. The Last Mile: High-Assurance and High-Speed Cryptographic Implementations. *CoRR* abs/1904.04606 (2019). arXiv:1904.04606 http://arxiv.org/abs/1904.04606

Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On Subnormal Floating Point and Abnormal Timing. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May*

17-21, 2015. 623–639. https://doi.org/10.1109/SP.2015.44

Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. 2018. Towards Verified, Constant-time Floating Point Operations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 1369–1382. https://doi.org/10.1145/3243734.3243766

Andrew W. Appel. 2011. Verified Software Toolchain - (Invited Talk). In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

Andrew W. Appel. 2014. *Program Logics - for Certified Compilers*. Cambridge University Press. http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/program-logics-certified-compilers?format=HB

Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.* 37, 2 (2015), 7:1–7:31. https://doi.org/10.1145/2701415

ARM. 2016. mbed TLS. https://tls.mbed.org/

Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. 2014. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. 328–343. https://doi.org/10.1109/CSF.2018.00031

Gilles Barthe, Tamara Rezk, and David A. Naumann. 2006. Deriving an Information Flow Checker and Certifying Compiler for Java. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*. IEEE Computer Society, 230–242. https://doi.org/10.1109/SP.2006.13

Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 207–221. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer

Daniel J Bernstein. 2006. Curve25519: new Diffie-Hellman speed records. In *International Workshop on Public Key Cryptography*. Springer, 207–228.

Daniel J Bernstein, Tanja Lange, and Peter Schwabe. 2012. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America*. Springer, 159–176.

Frédéric Besson, Alexandre Dang, and Thomas Jensen. 2019. Information-Flow Preservation in Compiler Optimisations. In *CSF*. IEEE.

Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. 2006. Formal Verification of a C Compiler Front-End. In *FM 2006 (LNCS)*, Vol. 4085. 460–475.

Sandrine Blazy, David Pichardie, and Alix Trieu. 2019. Verifying constant-time implementations by abstract interpretation. *Journal of Computer Security* 27, 1 (2019), 137–163. https://doi.org/10.3233/JCS-181136

Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: a DSL for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019.*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 174–189. https://doi.org/10.1145/3314221.3314605

Juan Chen, Ravi Chugh, and Nikhil Swamy. 2010. Type-preserving compilation of end-to-end verification of security enforcement. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 412–423. https://doi.org/10.1145/1806596.1806643

Inria 2019. *The Coq proof assistant reference manual*. Inria. http://coq.inria.fr Version 8.9.1.

Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A Formally-Verified C Static Analyzer. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 247–259. https://doi.org/10.1145/2676726.2676966

Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. 2018. Crellvm: verified credible compilation for LLVM. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 631–645. https://doi.org/10.1145/3192366.3192377

Daniel Kästner, Jörg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. 2018. CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler. In *ERTS2 2018 - 9th European Congress Embedded Real-Time Software and Systems*. 3AF, SEE, SIE, Toulouse, France, 1–9. https://hal.inria.fr/hal-01643290

Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. 2016. 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*). 573–582.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 179–192. https://doi.org/10.1145/2535838.2535841

Adam Langley. 2010. ctgrind. https://github.com/agl/ctgrind

Adam Langley. 2015. curve25519-donna. https://code.google.com/archive/p/curve25519-donna

Xavier Leroy. 2006. Formal certification of a compiler back-end or : Programming a compiler with a proof assistant. *POPL* (2006), 42–54.

Xavier Leroy. 2009a. Formal verification of a realistic compiler. *Commun. ACM* (2009).

Xavier Leroy. 2009b. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446.

Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2014. The CompCert memory model. In *Program Logics for Certified Compilers*, Andrew W. Appel (Ed.). Cambridge University Press, 237–271. https://hal.inria.fr/hal-00905435

Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. SEE, Toulouse, France. https://hal.inria.fr/hal-01238879

Chang Liu, Michael Hicks, and Elaine Shi. 2013. Memory Trace Oblivious Program Execution. In *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013*. IEEE Computer Society, 51–65. https://doi.org/10.1109/CSF.2013.11

Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony C. J. Fox. 2019. Verified compilation on a verified processor. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019.*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 1041–1053. https://doi.org/10.1145/3314221.3314622

David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. 2005. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers (Lecture Notes in Computer Science)*, Dongho Won and Seungjoo Kim (Eds.), Vol. 3935. Springer, 156–168. https://doi.org/10.1007/11734727_14

Toby C. Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah. 2016. 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*. IEEE Computer Society, 417–431. https://doi.org/10.1109/CSF.2016.36

Van Chan Ngo, Mario Dehesa-Azuara, Matthew Fredrikson, and Jan Hoffmann. 2017. Verifying and Synthesizing Constant-Resource Implementations with Types. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 710–728. https://doi.org/10.1109/SP.2017.53

OpenSSL. 2019. OpenSSL. https://www.openssl.org/

Scott Owens, Michael Norrish, Ramana Kumar, Magnus O. Myreen, and Yong Kiam Tan. 2017. Verifying efficient function calls in CakeML. *PACMPL* 1, ICFP (2017), 18:1–18:27. https://doi.org/10.1145/3110262

Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work. *ACM Comput. Surv.* 51, 6, Article 125 (Feb. 2019), 36 pages. https://doi.org/10.1145/3280984

Nadia Polikarpova, Jean Yang, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Type-Driven Repair for Information Flow Security. *CoRR* abs/1607.03445 (2016). arXiv:1607.03445 http://arxiv.org/abs/1607.03445

Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified low-level programming embedded in F. *PACMPL* 1, ICFP (2017), 17:1–17:29. https://doi.org/10.1145/3110261

Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. 2016. Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 110–120. https://doi.org/10.1145/2892208.2892230

Eyal Ronen, Kenneth G. Paterson, and Adi Shamir. 2018. Pseudo Constant Time Implementations of TLS Are Only Pseudo Secure. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 1397–1414. https://doi.org/10.1145/3243734.3243775

Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (2013), 22:1–22:50. https://doi.org/10.1145/2487241.2487248

Robert Sison and Toby Murray. 2019. Verifying that a compiler preserves concurrent value-dependent information-flow security. In *International Conference on Interactive Theorem Proving (Lecture Notes in Computer Science)*. Springer-Verlag.

SUPERCOP. 2019. SUPERCOP. https://bench.cr.yp.to/supercop.html

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. 2016. A new verified compiler backend for CakeML. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 60–73. https://doi.org/10.1145/2951913.2951924

Filippo Del Tedesco, David Sands, and Alejandro Russo. 2016. Fault-Resilient Non-interference. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 401–416. https://doi.org/10.1109/CSF.2016.35

Klaus von Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. 2019. IODINE: Verifying Constant-Time Execution of Hardware. In *USENIX Security Symposium*. USENIX.

David J Wheeler and Roger M Needham. 1994. TEA, a tiny encryption algorithm. In *International Workshop on Fast Software Encryption*. Springer, 363–366.

Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 15–26. https://doi.org/10.1145/3213846.3213851

Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. 2017. Verified Correctness and Security of mbedTLS HMAC-DRBG. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2007–2020. https://doi.org/10.1145/3133956.3133974

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 427–440. https://doi.org/10.1145/2103656.2103709

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2013. Formal verification of SSA-based optimizations for LLVM. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 175–186. https://doi.org/10.1145/2491956.2462164

Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1789–1806. https://doi.org/10.1145/3133956.3134043