Verifying Constant-Time Implementations by Abstract Interpretation

Sandrine Blazy[1(✉)], David Pichardie[2(✉)], and Alix Trieu[1(✉)]

[1] CNRS IRISA - Université Rennes 1 - Inria, Rennes, France
sandrine.blazy@irisa.fr, alix.trieu@irisa.fr
[2] CNRS IRISA - ENS Rennes - Inria, Rennes, France
david.pichardie@irisa.fr

Abstract. Constant-time programming is an established discipline to secure programs against timing attackers. Several real-world secure C libraries such as NaCl, mbedTLS, or Open Quantum Safe, follow this discipline. We propose an advanced static analysis, based on state-of-the-art techniques from abstract interpretation, to report time leakage during programming. To that purpose, we analyze source C programs and use full context-sensitive and arithmetic-aware alias analyses to track the tainted flows.

We give semantic evidences of the correctness of our approach on a core language. We also present a prototype implementation for C programs that is based on the CompCert compiler toolchain and its companion Verasco static analyzer. We present verification results on various real-world constant-time programs and report on a successful verification of a challenging SHA-256 implementation that was out of scope of previous tool-assisted approaches.

1 Introduction

To protect their implementations, cryptographers follow a very strict programming discipline called constant-time programming. They avoid branchings controlled by secret data as an attacker could use timing attacks, which are a broad class of side-channel attacks that measure different execution times of a program in order to infer some of its secret values [1,11,18,23]. They also avoid memory load/store indexed by secret data because of cache-timing attacks. Several real-world secure C libraries such as NaCl [7], mbedTLS [26], or Open Quantum Safe [30], follow this programming discipline.

The constant-time programming discipline requires to transform programs. These transformations may be tricky and error-prone, mainly because they involve low-level features of C and non-standard operations (e.g. bit-level manipulations). We argue that programmers need tool assistance to use this programming discipline. First, they need feedback at the source level during programming, in order to verify that their implementation is constant time and also to understand why a given implementation is not constant time as expected. Moreover, they need to trust that their compiler will not break source security

S.N. Foley et al. (Eds.): ESORICS 2017, Part I, LNCS 10492, pp. 260–277, 2017.
DOI: 10.1007/978-3-319-66402-6_16

when translating the guarantees obtained at the source level. Indeed, compiler optimizations could interfere with the previous constant-time transformations performed by the programmer. In this paper, we choose to implement static analysis at source level to simplify error reporting, but couple the analyzer to the highly trustworthy CompCert compiler [25]. This strategic design choice allows us to take advantage of static analysis techniques that would be hard to apply at lowest program representation levels.

Static analysis is frequently used for identifying security vulnerabilities in software, for instance to detect security violations pertaining to information flow [15,21,34]. In this paper, we propose an advanced static analysis, based on state-of-the-art techniques from abstract interpretation [12] (mainly fixpoint iterations operating over source programs, use of widening operators, computations performed by several abstract domains including a memory abstract domain handling pointer arithmetic), to report time leakage during programming.

Data originating from a statement where information may leak is tainted with the lowest security level. Our static analysis uses two security levels, that we call secret (high level) and public (low level); it analyzes source C programs and uses full context-sensitive (i.e., the static analysis distinguishes the different invocations of a same function) and arithmetic-aware alias analyses (i.e., the cells of an array are individually analyzed, even if they are accessed using pointer dereferencing and pointer arithmetic) to track the tainted flows.

We follow the abstract interpretation methodology: we design an abstract interpreter that executes over security properties instead of concrete values, and use approximation of program executions to perform fixpoint computations. We hence leverage the inference capabilities of advanced abstract interpretation techniques as relational numeric abstractions [28], abstract domain collaborations [19], arithmetic-aware alias analysis [9,27], to build a very precise taint analysis on C programs. As a consequence, even if a program uses a same memory block to store both secret and public values during computations, our analysis should be able to track it, without generating too many spurious false alarms. This programming pattern appears in real-world implementations, such as the SHA-256 implementation in NaCl that we are able to analyze.

In this paper, we make the following contributions:

- We define a new methodology for verifying constant-time security of C programs. Our static analysis is fully automatic and sound by construction.
- We instrument our approach in the Verasco static analyzer [22]. Verasco is a formally-verified static analyzer, that is connected to the formally-verified CompCert C compiler. We thus benefit from the CompCert correctness theorem, stating roughly that a compiled program behaves as prescribed by the semantics of its source program.
- We report our results obtained from a benchmark of representative cryptographic programs that are known to be constant time. Thanks to the precision of our static analyzer, we are able to analyze programs that are out of reach of state-of-the-art tools.

This paper is organized as follows. First, Sect. 2 presents the Verasco static analyzer. Then, Sect. 3 explains our methodology and details our abstract interpreter. Section 4 describes the experimental evaluation of our static analyzer. Related work is described in Sect. 5, followed by conclusions.

2 The Verasco Abstract Interpreter

Verasco is a static analyzer based on abstract interpretation that is formally verified in Coq [22]. Its proof of correctness ensures the absence of runtime errors (such as out-of-bound array accesses, null pointer dereference, and arithmetic exceptions) in the analyzed C programs. Verasco relies on several abstract domains, including a memory domain that finely tracks properties related to memory contents, taking into account type conversions and pointer arithmetic [9].

Verasco is connected to the CompCert formally-verified C compiler, that is also formally verified in Coq [25]. Its correctness theorem is a semantics preservation theorem; it states that the compilation does not introduce bugs in compiled programs. More precisely, Verasco operates over C#minor, a C-like language that is the second intermediate language in the CompCert compilation pipeline.

Verasco raises an alarm as soon as it detects a potential runtime error. Its correctness theorem states that if Verasco returns no alarm, then the analyzed program is *safe* (i.e., none of its observable behaviors is an undefined behavior, according to the C#minor semantics). The design of Verasco is inspired by Astrée [8], a milestone analyzer that was able to successfully analyze realistic safety-critical software systems for aviation and space flights. Verasco follows a similar modular architecture as Astrée, that is shown in Fig. 1.

First, at the bottom of the figure, a large hub of numerical abstract domains is provided to infer numerical invariants on programs. These properties can be *relational* as for example $j+1 \leq i \leq j+2$ in a loop (with *Octagons* or *Polyhedra* abstract domains). All these domains finely analyze the behavior of machine integers and floating-points (with potential overflows) while unsound analyzers would assume ideal arithmetic. They are connected all-together via *communication channels* that allow each domain to improve its own precision via specific queries to other domains. As a consequence, Verasco is able to infer subtle numerical invariants that require complex reasoning about linear arithmetic, congruence and symbolic equalities.

Second, on top of these numerical abstractions sits an abstract memory functor [9] that tracks fine-grained aliases and interacts with the numerical domains. This functor can choose to represent each cell of a same memory block with a single property, or to finely track each specific property of every position in the block. Contrary to many other alias analyses, this approach allows us to reason on local and global variables with the same level of precision, even when the memory addresses are manipulated by the programmer. Some unavoidable approximations are performed when the target of a memory dereference corresponds to several possible targets, but Verasco makes the impact of such imprecision as limited as possible. Because of ubiquitous pointer arithmetic in C programs (even simple

array accesses are represented via pointer arithmetic in C semantics), the functor needs to ask advanced symbolic numerical queries to the abstract numerical domain below it. In return, its role is to hide the load and store operations from them, and only communicate via symbolic numerical variables.

Third, the last piece of the analyzer is an advanced abstract interpreter that builds a fixpoint for the analysis result. This task is a bit more complex than in standard dataflow analysis techniques that look for the least solution of dataflow equation systems. In such settings, each equation is defined by means of monotone operators in a well chosen lattice without infinite ascending chains. By computing the successive iterates of the transfer functions attached to each equations, starting from a bottom element, the fixpoint computation always terminates on the least element of the lattice that satisfies all equations. In contrast, the Verasco abstract interpreter relies on infinite lattices, where widening and narrowing operators [12] are used for ensuring and accelerating the convergence. Smart iteration strategies are crucial when using such accelerating operators because they directly impact the precision of the analysis diagnosis. Verasco builds its strategy by following the structure of the program. On every program loop, it builds a local fixpoint using accelerating techniques. At every function call, it makes a recursive call of the abstract interpreter on the body of the callee. The callee may be resolved thanks to the state abstraction functor in presence of function pointers. The recursive nature of the abstract interpreter makes the analysis very precise because each function is independently analyzed as many times as there are calling contexts that invoke it.

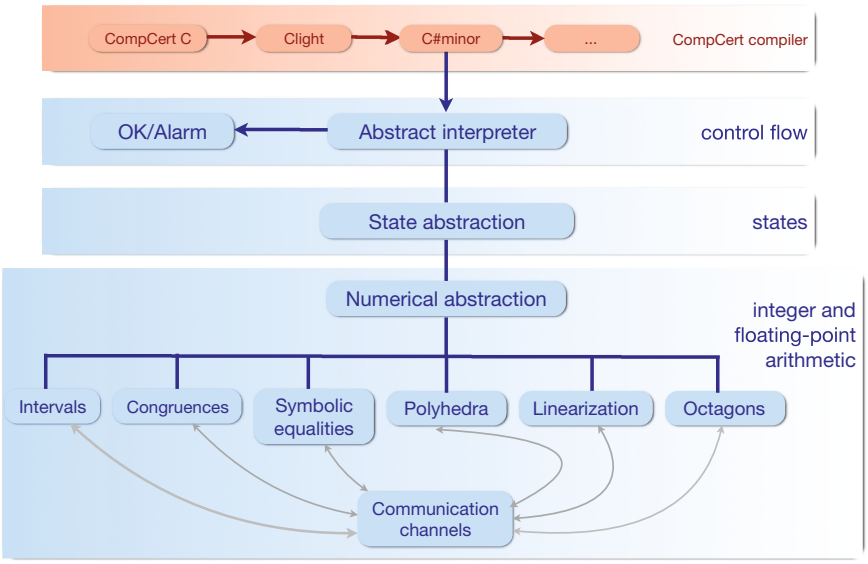

Fig. 1. Architecture of the Verasco static analyzer

Furthermore, C#minor is classically structured in functions, statements, and expressions. Expressions have no side effects; they include reading temporary variables (which do not reside in memory), taking the address of a non-temporary variable, constants, arithmetic operations, and dereferencing addresses. The arithmetic, logical, comparison, and conversion operators are roughly those of C, but without overloading: for example, distinct operators are provided for integer multiplication and floating-point multiplication. Likewise, there are no implicit casts: all conversions between numerical types are explicit. Statements offer both structured control and `goto` with labels. C loops as well as `break` and `continue` statements are encoded as infinite loops with a multi-level `exit` n that jumps to the end of the $(n+1)$-th enclosing `block`.

3 Verifying Constant-Time Security

Our static analyzer operates over C#minor programs. In this paper, we use a simpler While toy-language for clarity. It is defined in the first part of this section. Then, we detail our model for constant-time leakage, and explain the tainting semantics we have defined to track data dependencies in programs. Last, we explain the main algorithm of our static analyzer.

3.1 The While Language

Our While language is classically structured in statements and expressions, as shown in Fig. 2. Expressions include constants, addresses of variables, arithmetic operations and dereferencing addresses, so as to model pointer aliasing. Statements include skip statements, stores, sequences, if and while statements.

Expressions: $e ::= n \mid x \mid e_1 \oplus e_2 \mid *e$

Statements: $p ::= \texttt{skip} \mid e_1 = e_2 \mid p_1 ; p_2 \mid \texttt{if } e \texttt{ then } p_1 \texttt{ else } p_2 \mid \texttt{while } e \texttt{ do } p$

Fig. 2. Syntax of While programs

The semantics of While is defined in Fig. 3 using a small-step style, supporting the reasoning on nonterminating programs. Contrary to the C language, the semantics is deterministic (and so is the semantics of C#minor). Given a semantic state σ, an expression e evaluates (big-step style) in a value v (written $\langle \sigma, e \rangle \rightarrow v$); the execution of a statement s results in an updated state σ' and a new statement to execute s' (written $\langle \sigma, s \rangle \rightarrow \langle \sigma', s' \rangle$). The semantic state σ maps memory locations (pairs $l = (x, n)$ made of an address and an offset from this address) to values. Values can either be locations or constants, and we write $\sigma(e)$ to denote the value of expression e in state σ (i.e. $\langle \sigma, e \rangle \rightarrow \sigma(e)$).

The reflexive transitive closure of this small-step semantics represents the execution of a program. When the program terminates (resp. diverges, e.g. when

$$\frac{}{\langle \sigma, n\rangle \to n} \qquad \frac{}{\langle \sigma, x\rangle \to (x, 0)}$$

$$\frac{\langle \sigma, e_1\rangle \to l \quad \sigma(l) = v}{\langle \sigma, *e_1\rangle \to v} \qquad \frac{\langle \sigma, e_1\rangle \to v_1 \quad \langle \sigma, e_2\rangle \to v_2}{\langle \sigma, e_1 \oplus e_2\rangle \to v_1 \oplus v_2}$$

$$\frac{\langle \sigma, e_1\rangle \to l \quad \langle \sigma, e_2\rangle \to n}{\langle \sigma, e_1 = e_2\rangle \to \langle \sigma[l \mapsto n], \texttt{skip}\rangle}$$

$$\frac{}{\langle \sigma, \texttt{skip; } p\rangle \to \langle \sigma, p\rangle} \qquad \frac{\langle \sigma, p_1\rangle \to \langle \sigma', p_1'\rangle}{\langle \sigma, p_1\texttt{; } p_2\rangle \to \langle \sigma', p_1'\texttt{; } p_2\rangle}$$

$$\frac{\langle \sigma, e\rangle \to \texttt{true}}{\langle \sigma, \texttt{if } e \texttt{ then } p_1 \texttt{ else } p_2\rangle \to \langle \sigma, p_1\rangle} \qquad \frac{\langle \sigma, e\rangle \to \texttt{false}}{\langle \sigma, \texttt{if } e \texttt{ then } p_1 \texttt{ else } p_2\rangle \to \langle \sigma, p_2\rangle}$$

$$\frac{\langle \sigma, e\rangle \to \texttt{true}}{\langle \sigma, \texttt{while } e \texttt{ do } p\rangle \to \langle \sigma, p\texttt{; while } e \texttt{ do } p\rangle} \qquad \frac{\langle \sigma, e\rangle \to \texttt{false}}{\langle \sigma, \texttt{while } e \texttt{ do } p\rangle \to \langle \sigma, \texttt{skip}\rangle}$$

Fig. 3. Semantics of While programs

an infinite loop is executed), it is a finite (resp. infinite) execution of steps. The execution of a program is *safe* iff either the program diverges, or the program terminates (i.e., its final semantic state is $\langle \sigma, \texttt{skip}\rangle$, meaning that there is no more statement to execute). The execution of a program is *stuck* (on $\langle \sigma, s\rangle$) when s differs from `skip` and no semantic rule can be applied.

3.2 Constant-Time Security

In our model, we assume that branching statements and memory accesses may leak information through their execution. We use a similar definition of constant-time security to the one given in [2]. We define a *leakage model* $\mathtt{L}$ as a map from semantic states $\langle \sigma, p\rangle$ to sequences of observations $\mathtt{L}(\langle \sigma, p\rangle)$ with ε being the empty observation. Two executions are said to be *indistinguishable* when their observations are the same:

$$\mathtt{L}(\langle \sigma_0, p_0\rangle) \cdot \mathtt{L}(\langle \sigma_1, p_1\rangle) \cdot \ldots = \mathtt{L}(\langle \sigma_0', p_0'\rangle) \cdot \mathtt{L}(\langle \sigma_1', p_1'\rangle) \cdot \ldots.$$

Definition 1 (Constant-time leakage model). *Our leakage model is such that the three following equalities hold, where $*e_0', \ldots * e_n'$ are the read memory accesses appearing respectively in expressions e in the first line, e in the second line, e_1 and e_2 in the third line.*

1. $\mathtt{L}(\langle \sigma, \texttt{if } e \texttt{ then } p_1 \texttt{ else } p_2\rangle) = \sigma(e)\sigma(e_0') \ldots \sigma(e_n')$
2. $\mathtt{L}(\langle \sigma, \texttt{while } e \texttt{ do } p\rangle) = \sigma(e)\sigma(e_0') \ldots \sigma(e_n')$
3. $\mathtt{L}(\langle \sigma, e_1 = e_2\rangle) = \sigma(e_1)\sigma(e_0') \ldots \sigma(e_n')$

The first and second lines mean that the value of branching conditions is considered as leaked. The last line means that the address of a store access is also considered as leaked. Additionally, all locations of read accesses are also considered as leaked.

Given this leakage model, two indistinguishable executions of a program must necessarily have the same control flow. Moreover, one execution cannot be stuck while the other can continue execution. Indeed, in our While language, the only way to have a stuck execution is either to try to dereference a value that is not a valid location (a constant or an out-of-range location), or to write in a constant value or to branch on a non-boolean value. However, by definition of indistinguishability and the leakage model, these values must be the same in both executions, thus both executions have the same control flow.

Given a program, we assume that the attacker has access to the values of some of its inputs, which we call the *public* input variables, and does not have access of the other ones, which we call the *secret* input variables.

Definition 2 (Constant-time security). *A program p_0 is constant time if for any set X_i of public input variables such that for all pair of safe executions $\langle \sigma_0, p_0 \rangle \rightarrow \langle \sigma_1, p_1 \rangle \rightarrow \dots$ and $\langle \sigma'_0, p_0 \rangle \rightarrow \langle \sigma'_1, p'_1 \rangle \rightarrow \dots$ such that both states share the same public values (i.e., $\forall x \in X_i, i \in \mathbb{N}, \sigma_0(x,i) = \sigma'_0(x,i)$), then both executions are indistinguishable.*

This definition means that a constant-time program is such that, any pair of its executions that only differ on its secrets must leak the exact same information. This also gives a definition of constant-time security for infinite execution.

3.3 Reducing Security to Safety

We introduce an intermediate tainting semantics for While programs in Fig. 4, and use the $\rightsquigarrow$ symbol to distinguish its executions from those of the original semantics. The tainting semantics is an instrumentation of the While semantics that tracks dependencies. In the tainted semantics, a program gets stuck if branchings or memory accesses depend on secrets. We introduce taints, either `High` or `Low` to respectively represent secret and public values and a union operator on taints defined as follows: $\texttt{Low} \sqcup \texttt{Low} = \texttt{Low}$ and $\forall\ \texttt{t}, \texttt{High} \sqcup \texttt{t} = \texttt{t} \sqcup \texttt{High} = \texttt{High}$; it is used to compute the taint of a binary expression. In the instrumented semantics, we take into account taints in semantic values: the semantic state σ becomes a tainted state σ_τ, where locations are now mapped to pairs made of a value and a taint.

Let us note that for a dereferencing expression $*e_1$ to have a value, the taint associated to e_1 must be `Low`. Indeed, we forbid memory read accesses that might leak secret values. This concerns dereferencing expressions (loads) and assignment statements (store of a lvalue). Similarly, test conditions in branching statements must also have a `Low` taint.

The instrumented semantics preserves the regular behavior of programs (defined in Fig. 3), as stated by the following theorem, which can be proven by induction on the execution relation.

Theorem 1. *Any safe execution $\langle \sigma_{\tau 0}, p_0 \rangle \rightsquigarrow \langle \sigma_{\tau 1}, p_1 \rangle \rightsquigarrow \dots$ of program p_0 in the tainting semantics implies that the execution $\langle \sigma_0, p_0 \rangle \rightarrow \langle \sigma_1, p_1 \rangle \rightarrow \dots$ is also safe in the regular semantics. Here, for all k, σ_k is a semantic state such*

$$\frac{}{\langle \sigma_\tau, n\rangle \rightsquigarrow (n, \texttt{Low})} \qquad \frac{}{\langle \sigma_\tau, x\rangle \rightsquigarrow ((x,0), \texttt{Low})}$$

$$\frac{\langle \sigma_\tau, e_1\rangle \rightsquigarrow (l, \texttt{Low}) \quad \sigma_\tau(l) = (v,t)}{\langle \sigma_\tau, *e_1\rangle \rightsquigarrow (v,t)} \qquad \frac{\langle \sigma_\tau, e_1\rangle \rightsquigarrow (v_1,t_1) \quad \langle \sigma_\tau, e_2\rangle \rightsquigarrow (v_2,t_2)}{\langle \sigma_\tau, e_1 \oplus e_2\rangle \rightsquigarrow (v_1 \oplus v_2, t_1 \sqcup t_2)}$$

$$\frac{\langle \sigma_\tau, e_1\rangle \rightsquigarrow (l, \texttt{Low}) \quad \langle \sigma_\tau, e_2\rangle \rightsquigarrow (n,t)}{\langle \sigma_\tau, e_1 = e_2\rangle \rightsquigarrow \langle \sigma_\tau[l \mapsto (n,t)], \texttt{skip}\rangle}$$

$$\frac{}{\langle \sigma_\tau, \texttt{skip; } p\rangle \rightsquigarrow \langle \sigma_\tau, p\rangle} \qquad \frac{\langle \sigma_\tau, p_1\rangle \rightsquigarrow \langle \sigma'_\tau, p'_1\rangle}{\langle \sigma_\tau, p_1\texttt{; } p_2\rangle \rightsquigarrow \langle \sigma'_\tau, p'_1\texttt{; } p_2\rangle}$$

$$\frac{\langle \sigma_\tau, e\rangle \rightsquigarrow (\texttt{true}, \texttt{Low})}{\langle \sigma_\tau, \texttt{if } e \texttt{ then } p_1 \texttt{ else } p_2\rangle \rightsquigarrow \langle \sigma_\tau, p_1\rangle} \qquad \frac{\langle \sigma_\tau, e\rangle \rightsquigarrow (\texttt{false}, \texttt{Low})}{\langle \sigma_\tau, \texttt{if } e \texttt{ then } p_1 \texttt{ else } p_2\rangle \rightsquigarrow \langle \sigma_\tau, p_2\rangle}$$

$$\frac{\langle \sigma_\tau, e\rangle \rightsquigarrow (\texttt{true}, \texttt{Low})}{\langle \sigma_\tau, \texttt{while } e \texttt{ do } p\rangle \rightsquigarrow \langle \sigma_\tau, p\texttt{; while } e \texttt{ do } p\rangle} \qquad \frac{\langle \sigma_\tau, e\rangle \rightsquigarrow (\texttt{false}, \texttt{Low})}{\langle \sigma_\tau, \texttt{while } e \texttt{ do } p\rangle \rightsquigarrow \langle \sigma_\tau, \texttt{skip}\rangle}$$

Fig. 4. Tainting semantics for While programs

that for all location l where $\sigma_{\tau k}$ is defined, there exists a taint t_k such that $\sigma_{\tau k}(l) = (\sigma_k(l), t_k)$. As an immediate corollary, any safe program according to the tainting semantics is also safe according to the regular semantics.

Theorem 1 is useful to prove our main theorem relating our instrumented semantics and the constant-time property we want to verify on programs.

Theorem 2. *Any safe program w.r.t. the tainting semantics is constant time.*

Proof. Let p_0 be a safe program with respect to the tainting semantics. Let X_i be a set of public variables and let $\langle \sigma_0, p_0\rangle \rightarrow \langle \sigma_1, p_1\rangle \rightarrow \ldots$ and $\langle \sigma'_0, p_0\rangle \rightarrow \langle \sigma'_1, p'_1\rangle \rightarrow \ldots$ be two safe executions of p_0 such that for all $x \in X_i$ and $n \in \mathbb{N}$, we have $\sigma_0(x,n) = \sigma'_0(x,n)$.

We now need to prove that both executions are indistinguishable. Let $\sigma_{\tau 0}$ be such that for all $x \in X_i,\ n \in \mathbb{N},\ \sigma_{\tau 0}(x,n) = (\sigma_0(x,n), \texttt{Low})$ and also for all $x \notin X_i,\ n \in \mathbb{N},\ \sigma_{\tau 0}(x,n) = (\sigma_0(x,n), \texttt{High})$.

By safety of program p_0 according to the tainting semantics, there exists some states $\sigma_{\tau 1}, \sigma_{\tau 2}, \ldots$ such that $\langle \sigma_{\tau 0}, p_0\rangle \rightsquigarrow \langle \sigma_{\tau 1}, p_1\rangle \rightsquigarrow \ldots$ is a safe execution. Let $\sigma_{n'}$ be such that there exists for all location l, a taint t'_n such that, $\sigma_{\tau n}(l) = (\sigma_{n'}(l), t'_n)$. We prove by strong induction on n that $\sigma_{n'} = \sigma_n$.

- It is clearly true for $n = 0$ by definition of $\sigma_{\tau 0}$.
- Suppose it is true for all $k < n$ and let us prove it for n. By using Theorem 1, we know that there exists a safe execution $\langle \sigma_0, p_0\rangle \rightarrow \langle \sigma_{1'}, p_{1'}\rangle \rightarrow \ldots \rightarrow \langle \sigma_{n'}, p_{n'}\rangle \rightarrow \ldots$. Furthermore, the semantics is deterministic and we know that $\langle \sigma_0, p_0\rangle \rightarrow \langle \sigma_1, p_1\rangle \rightarrow \ldots$. Therefore, we have the following series of equalities: $\sigma_{1'} = \sigma_1, p_{1'} = p_1, \ldots \sigma_{n'} = \sigma_n, p_{n'} = p_n$.

Thus, for all $k \in \mathbb{N}$, the state $\sigma_{\tau k}$ verifies that for all l, there exists t_k such that $\sigma_{\tau k}(l) = (\sigma_k(l), t_k)$. Similarly, we define $\sigma'_{\tau 0}, \sigma'_{\tau 1}, \ldots$ for the second execution which also verifies the same property by construction.

Finally, we need to prove that for all $n \in \mathbb{N}$, $L(\langle \sigma_n, p_n \rangle) = L(\langle \sigma'_n, p'_n \rangle)$. First, we define the notation $\sigma_n =_{\mathtt{L}} \sigma'_n$ for all $n \in \mathbb{N}$, meaning that for all l, $\sigma_{\tau n}(l) = (\sigma_n(l), \mathtt{Low})$ iff $\sigma'_{\tau n}(l) = (\sigma'_n(l), \mathtt{Low})$ and $\sigma_n(l) = \sigma'_n(l)$. Both environments must agree on locations with `Low` taints. For all $n \in \mathbb{N}$, let us prove by induction on p_n that if $p_n = p'_n$ and $\sigma_n =_{\mathtt{L}} \sigma'_n$, then $p_{n+1} = p'_{n+1}$ and $\sigma_{n+1} =_{\mathtt{L}} \sigma'_{n+1}$.

- if $p_n = \mathtt{skip;}p'$, it is true because $p_{n+1} = p'_{n+1} = p'$, $\sigma_{n+1} = \sigma_n$ and also $\sigma'_{n+1} = \sigma'_n$.
- if $p_n = p\mathtt{;}\ p'$, it is true by induction hypothesis.
- if $p_n = \mathtt{if}\ e \ldots$ or $p_n = \mathtt{while}\ e \ldots$, we have $\sigma_{n+1} = \sigma_n$ and $\sigma'_{n+1} = \sigma'_n$. Furthermore, we know that there exists some v such that $\langle \sigma_{\tau n}, e \rangle \rightsquigarrow (v, \mathtt{Low})$ and similarly, there exists v' such that $\langle \sigma'_{\tau n}, e \rangle \rightsquigarrow (v', \mathtt{Low})$ because of the safety in the tainting semantics. Since $\sigma_n(e) = v$, $\sigma'_n(e) = v'$ and $\sigma_n =_{\mathtt{L}} \sigma'_n$, we have $v = v'$ and thus $p_{n+1} = p'_{n+1}$.
- if $p_n = e_1 = e_2$, we have $p_{n+1} = p'_{n+1} = \mathtt{skip}$. By using the same reasoning as the previous case, we can prove that $\sigma_n(e_1) = \sigma'_n(e_1) = l$. There exists v, v', t, t' such that $\langle \sigma_{\tau n}, e_2 \rangle \rightsquigarrow (v, t)$ and $\langle \sigma'_{\tau n}, e_2 \rangle \rightsquigarrow (v', t')$ and thus $\sigma_{n+1} = \sigma_n[l \mapsto v]$ and $\sigma'_{n+1} = \sigma_n[l \mapsto v']$. If $t = t' = \mathtt{Low}$, then $v = v'$ and $\sigma_{n+1} =_{\mathtt{L}} \sigma'_{n+1}$. If $t = t' = \mathtt{High}$, then $\sigma_{n+1} =_{\mathtt{L}} \sigma'_{n+1}$ by definition. Without loss of generality, we can consider that $t = \mathtt{Low}$ and $t' = \mathtt{High}$. e_2 must necessarily contain a memory read $*e_3$ such that $\langle \sigma'_{\tau n}, *e_3 \rangle \rightsquigarrow (v_3, \mathtt{High})$ otherwise $t' = \mathtt{Low}$. With a similar reasoning than before, we can prove $\sigma_n(e_3) = \sigma'_n(e_3) = l'$. So, $\sigma_{\tau n}(l')$ must have `High` taint by definition of $\sigma_n =_{\mathtt{L}} \sigma'_n$, which is absurd.

Finally, by exploiting this lemma, a simple induction proves that for all $n \in \mathbb{N}$, $p_n = p'_n$ and $\sigma_n =_{\mathtt{L}} \sigma'_n$. Furthermore, a direct consequence is that for all $n \in \mathbb{N}$, $L(\langle \sigma_n, p_n \rangle) = L(\langle \sigma'_n, p'_n \rangle)$.

3.4 Abstract Interpreter

To prove that a program is safe according to the tainting semantics, we design a static analyzer based on abstract interpretation. It computes a correct approximation of the execution the analyzed program, thus if the approximative execution is safe, then the actual execution must necessarily be safe.

As our actual implementation takes advantage of the Verasco static analyzer, we reuse its memory abstraction $M^\#$. It provides $\mathtt{target}^\#$, $\mathtt{assert}^\#$ and $\mathtt{assign}^\#$ operators working as follows. Given an abstract environment $\sigma^\#$ and an expression e, $\mathtt{target}^\#(e, \sigma^\#)$ returns a list of locations $l_1, \ldots l_n$ corresponding to the locations that are represented by e. It returns $\bot^\#$ if e cannot be evaluated to a location. Second, suppose that we have an if statement with condition $(*x < 5)$ and the abstract environment only knows that location $(x, 0)$ has its value in $[0, 42]$. The analysis can gain precision by assuming in the first (resp. second) branch that location $(x, 0)$ has its value in $[0, 4]$ (resp. $[5, 42]$). Similarly, if we know that $(x, 0)$ only has its value between in $[-2, 4]$, we do not need to analyze the second branch. Thus, given an abstract environment $\sigma^\#$ and an expression e, $\mathtt{assert}^\#(e, \sigma^\#)$ returns a modified abstract environment assuming

that e is true; if it is not possible (because e can only evaluate to false in $\sigma^\#$ for example), it returns the error state $\bot^\#$. Third, $\texttt{assign}^\#(e_1, e_2, \sigma^\#)$ is the abstract counterpart of $e_1 = e_2$.

Now, for the analysis to track taints, we need an abstraction of taints $\texttt{Taint}^\#$ that we define as $\texttt{Low}^\#$ and $\texttt{High}^\#$. We use $\texttt{Low}^\#$ to indicate that a location contains a value that has exactly a `Low` taint and $\texttt{High}^\#$ to indicate that it may be `Low` or `High`. In order to analyze the following snippet, it is necessary to correctly approximate the taint that will be assigned to location $(x, 0)$ after execution.

```
if /* low expr */ x = /* high expr */ else x = /* low expr */
```

As it can either be `Low` or `High`, we use the approximation $\texttt{High}^\#$. We could have used $\texttt{High}^\#$ to indicate that a location can only have a `High` value, however constant-time security is not interested in knowing that value has exactly `High` taint, but only in knowing that it *may* have a `High` taint.

The analyzer is now given a new mapping $\tau^\#$ that maps locations to abstract taints. Given $\sigma^\#$, $\tau^\#$ and an expression e, we define a new operator $\texttt{low}(e, \sigma^\#, \tau^\#)$ asserting that e has `Low` taint and contains only non-secret dependent memory reads. It is defined recursively as follows. The tricky part is $*e$, where the operator verifies that e has a low taint to ensure that the memory access is not secret dependent and then uses $\texttt{targets}^\#$ to ensure that all possible accessed locations contain `Low` values.

- $\texttt{low}(n, \sigma^\#, \tau^\#) = \texttt{true}$ and $\texttt{low}(x, \sigma^\#, \tau^\#) = \texttt{true}$
- $\texttt{low}(*e, \sigma^\#, \tau^\#) = \texttt{low}(e, \sigma^\#, \tau^\#) \wedge \bigwedge_{l_i \in \texttt{targets}^\#(e,\sigma^\#)} (\tau^\#(l_i) = \texttt{Low}^\#)$
- $\texttt{low}(e_1 \oplus e_2, \sigma^\#, \tau^\#) = \texttt{low}(e_1, \sigma^\#, \tau^\#) \wedge \texttt{low}(e_2, \sigma^\#, \tau^\#)$

Similarly to `low`, we define $\texttt{safe}(e, \sigma^\#, \tau^\#)$ as asserting that e does not contain secret dependent memory reads but does not check the taint of e. We also define $\texttt{taint}^\#(e, \sigma^\#, \tau^\#)$ as the abstract taint of expression e. Moreover, to take account of taintings, we then define $\texttt{assert}^\#_\tau$ and $\texttt{assign}^\#_\tau$ as follows.

$$\texttt{assert}^\#_\tau\ (e, \sigma^\#, \tau^\#) = \texttt{if}\ \texttt{low}(e, \sigma^\#, \tau^\#)\ \texttt{then}\ (\texttt{assert}^\#(e, \sigma^\#), \tau^\#)\ \texttt{else}\ \bot^\#$$

$$\texttt{assign}^\#_\tau\ (e_1, e_2, \sigma^\#, \tau^\#) = \texttt{if}\ \texttt{low}(e_1, \sigma^\#, \tau^\#)\ \wedge\ \texttt{safe}(e_2, \sigma^\#, \tau^\#)\ \texttt{then}$$

$$(\texttt{assign}^\#(e_1, e_2, \sigma^\#), \bigsqcup_{l_i \in \texttt{targets}(e,\sigma^\#)} \tau^\#[l_i \mapsto \texttt{taint}^\#(e, \sigma^\#, \tau^\#)])\ \texttt{else}\ \bot^\#.$$

Finally, the abstract analysis $[\![p]\!](\sigma^\#, \tau^\#)$ of program p starting with abstract environment $\sigma^\#$ and tainting $\tau^\#$ is defined in Fig. 5. To analyze $(p_1\ ;\ p_2)$, first p_1 is analyzed and then p_2 is analyzed using the environment given by the first analysis. Similarly, to analyze a statement $(\texttt{if}\ e\ \texttt{then}\ p_1\ \texttt{else}\ p_2)$, p_1 is analyzed assuming that e is true and p_2 is analyzed assuming the opposite, $\sqcup^\#$ is then used to get an over-approximation of both results.

The loop $(\texttt{while}\ e\ \texttt{do}\ p)$ is the trickiest part to analyze, as the analysis cannot just analyze one iteration of the loop body and then recursively analyze the loop again since it may never terminate. It thus tries to find a loop invariant.

$$
\begin{aligned}
[\![\texttt{skip}]\!]^{\#}(\sigma^{\#},\tau^{\#}) &= (\sigma^{\#},\tau^{\#}) \\
[\![e_1 = e_2]\!]^{\#}(\sigma^{\#},\tau^{\#}) &= \texttt{assign}^{\#}_{\tau}(e_1,e_2,\sigma^{\#},\tau^{\#}) \\
[\![p_1\texttt{;}\ p_2]\!]^{\#}(\sigma^{\#},\tau^{\#}) &= [\![p_2]\!]^{\#}([\![p_1]\!]^{\#}(\sigma^{\#},\tau^{\#})) \\
[\![\texttt{if}\ e\ \texttt{then}\ p_1\ \texttt{else}\ p_2]\!]^{\#}(\sigma^{\#},\tau^{\#}) &= [\![p_1]\!]^{\#}(\texttt{assert}^{\#}_{\tau}(e,\sigma^{\#},\tau^{\#})) \sqcup^{\#} \\
&\quad\ [\![p_2]\!]^{\#}(\texttt{assert}^{\#}_{\tau}(\texttt{not}\ e,\sigma^{\#},\tau^{\#})) \\
[\![\texttt{while}\ e\ \texttt{do}\ p]\!]^{\#}(\sigma^{\#},\tau^{\#}) &= \texttt{assert}^{\#}_{\tau}(\texttt{not}\ e,\texttt{pfp}(\texttt{iter}(e,p,\cdot),(\sigma^{\#},\tau^{\#}))) \\
\texttt{iter}(e,p,(\sigma^{\#},\tau^{\#})) &= (\sigma^{\#},\tau^{\#}) \sqcup^{\#} \texttt{assert}^{\#}_{\tau}(e,[\![p]\!]^{\#}(\sigma^{\#},\tau^{\#}))
\end{aligned}
$$

Fig. 5. Abstract execution of statements

The standard method in abstract interpretation is to compute a post-fixpoint of the function $\texttt{iter}(e,p,\cdot)$ as defined in Fig. 5. It represents a loop invariant, the final result is thus the invariant where the test condition does not hold anymore. In order to compute the post-fixpoint, we use $\texttt{pfp}(f,x)$ which computes a post-fixpoint of monotone function f by successively computing $x, f(x), f(f(x)), \ldots$, and forces convergence using a widening-narrowing operator on the $M^{\#}$ part. The taint part does not require convergence help because it is a finite lattice.

3.5 Correctness of the Abstract Interpreter

In order to state the correctness of our abstract interpreter, we introduce the concept of concretization. We use $v \in \gamma(v^{\#})$ to say that v is in the concretization of abstract value $v^{\#}$, which means that $v^{\#}$ represents a set of concrete values of which v is a member.

The abstract interpreter operates over a product $M^{\#} \times T^{\#}$ of abstract environments and abstract taintings (maps from location to taints). For $\sigma^{\#} \in M^{\#}$, we suppose we already have its concretization $\gamma_1(\sigma^{\#})$ (as given in [9]). For $\tau^{\#} \in T^{\#}$, we first define the concretization of abstract taints by $\gamma_\tau(\texttt{Low}^{\#}) = \{\texttt{Low}\}$ and $\gamma_\tau(\texttt{High}^{\#}) = \{\texttt{Low},\texttt{High}\}$. For all σ_τ, we call σ^1_τ and σ^2_τ the two functions such that for all l, $\sigma_\tau(l) = (\sigma^1_\tau(l), \sigma^2_\tau(l))$. The concretization $\gamma_2(\tau^{\#})$ is then defined as follows.

$$\gamma_2(\tau^{\#}) = \{\sigma^2_\tau | \forall l, \sigma^2_\tau(l) \in \gamma_\tau(\tau^{\#}(l))\}$$

Finally, for all $(\sigma^{\#},\tau^{\#}) \in M^{\#} \times T^{\#}$, its concretization $\gamma(\sigma^{\#},\tau^{\#})$ is defined as

$$\gamma(\sigma^{\#},\tau^{\#}) = \{\sigma_\tau | \sigma^1_\tau \in \gamma_1(\sigma^{\#}) \wedge \sigma^2_\tau \in \gamma_2(\tau^{\#})\}$$

The correctness theorem of the abstract interpreter intuitively means that if the abstract interpreter does not raise an alarm, then the program must be safe according to the tainting semantics (in which case it is also safe according to the original semantics, because of Theorem 1). The correctness theorem can be stated as follows.

Theorem 3. *For all program p, environment σ_τ and abstract environment $\sigma_\tau^\#$ such that $\sigma_\tau \in \gamma(\sigma_\tau^\#)$, if we have the execution $\langle \sigma_\tau, p\rangle \rightsquigarrow^* \langle \sigma'_\tau, \texttt{skip}\rangle$, then we also have $\sigma'_\tau \in \gamma([\![p]\!]^\#(\sigma_\tau^\#))$.*

In order to prove this theorem, we follow the usual methodology in abstract interpretation and define a *collecting* semantics, aiming at facilitating the proof. The semantics (not detailed in the paper) still expresses the dynamic behavior of programs but takes a closer form to the analysis. It operates over properties of concrete environments, thus bridging the gap between concrete environments and abstract environments, which represent sets of concrete environments.

4 Implementation and Experiments

Following the methodology presented in Sect. 3, we have implemented a prototype leveraging the Verasco static analyzer. We have been able to evaluate our prototype by verifying multiple actual C code constant-time algorithms taken from different cryptographic libraries such as NaCl [7], mbedTLS [26], Open Quantum Safe [10] and `curve25519-donna` [16].

In order to use our tool, the user simply has to indicate which variables are to be considered secrets and the prototype will either raise alarms indicating where secrets may leak, or indicate that the input program is constant time. The user can either indicate a whole global variable to be considered as secret at the start of the program, or uses the `verasco_any_int_secret` built-in function to produce a random signed integer to be considered as secret.

4.1 Memory Separation

By leveraging Verasco, the prototype has no problem handling difficult problems such as memory separation. For example, the small example of Fig. 6 is easily proved as constant time. In this program, an array `t` is initialized with random values, such that the values in odd offsets are considered as secrets, contrary to values in even offsets. So, the analyzer needs to be precise enough to distinguish between the array cells and to take into account pointer arithmetic. The potential leak happens on line 6. However, the condition on line 5 constrains `i%2 == 0` to be true, and thus `i` must be even on line 6, so `t[i]` does not contain a secret. A naive analyzer would taint the whole array as secret and would thus not be able to prove the program constant-time, however our prototype has no problem to prove it.

Interestingly, an illustration of the problem can be found in real-world programs. For example, the NaCl implementation of SHA-256 is not handled by [2] due to this. Indeed, in this program, the hashing function uses the following C `struct` as an internal state that contains both secret and public values during execution.

```
int main(void) {
  int t[4] = { verasco_any_int(), verasco_any_int_secret(),
       verasco_any_int(), verasco_any_int_secret() };
  for (int i = 0; i < 4; i++)
     if (i%2 == 0) {  // First if condition
       if (t[i])  t[i] = 0; } // Second if condition
  return 0; }
```

Fig. 6. An example program that is analyzed as constant time

```
typedef  struct  crypto_hash_sha256_state  {
  uint32_t       state [ 8 ];
  uint32_t       count [ 2 ];
  unsigned  char  buf [ 64 ];          }  crypto_hash_sha256_state ;
```

While field `count` contains public values, fields `state` and `buf` can contain both public and secret values. Only `count` is used in possibly leaking operations, however the whole `struct` is allocated as a single memory block at low level (i.e., LLVM) and [2] does not manage to prove the memory separation.

4.2 Cryptographic Algorithms

We report in Table 1 our results on a set of cryptographic algorithms, all executions times reported were obtained on a 3.1 GHz Intel i7 with 16 GB of RAM. Sizes are reported in terms of numbers of C#minor statements (i.e., close to C statements), lines of code are measured with `cloc` and execution times are reported in seconds. The first block of lines gathers test cases for the implementations of a representative set of cryptographic primitives including TEA [36], an implementation of sampling in a discrete Gaussian distribution by Bos et al. [10] (`rlwe_sample`) taken from the Open Quantum Safe library [30], an implementation of elliptic curve arithmetic operations over Curve25519 [6] by Langley [16](`curve25519-donna`), and various primitives such as AES, DES, etc. The second block reports on different implementations from the NaCl library [7]. The third block reports on implementations from the mbedTLS [26] library. Finally, the last result corresponds to an implementation of MAC-then-Encode-then-CBC-Encrypt (MEE-CBC).

All these examples are proven constant time, except for AES and DES. Our prototype rightfully reports memory accesses depending on secrets, so these two programs are not constant time. Similarly to [2], `rlwe_sample` is only proven constant time, provided that the core random generator is also constant time, thus showing that it is the only possible source of leakage.

The last example `mee-cbc` is a full implementation of the MEE-CBC construction using low-level primitives taken from the NaCl library. Our prototype is able to verify the constant-time property of this example, showing that it scales to large code bases (939 loc).

Table 1. Verification of cryptographic primitives

Example	Size	Loc	Time
`aes`	1171	1399	41.39
`curve25519-donna`	1210	608	586.20
`des`	229	436	2.28
`rlwe_sample`	145	1142	30.76
`salsa20`	341	652	0.04
`sha3`	531	251	57.62
`snow`	871	460	3.37
`tea`	121	109	3.47
`nacl_chacha20`	384	307	0.34
`nacl_sha256`	368	287	0.04
`nacl_sha512`	437	314	1.02
`mbedtls_sha1`	544	354	0.19
`mbedtls_sha256`	346	346	0.38
`nbedtls_sha512`	310	399	0.26
`mee-cbc`	1959	939	933.37

Our prototype is able to verify a similar set of programs as [2], except for the `libfixedtimefixedpoint` library [3] which unfortunately does not use standard C and is not handled by CompCert. The library uses extensively a GNU extension known as statement-expressions and would require heavy rewriting to be accepted by our tool.

On the other hand, our tool shows its agility with memory separation on the program SHA-256 that was out of reach for [2] and its restricted alias management. In terms of analysis time, our tool behaves similarly to [2]. On a similar experiment platform, we observe a speedup between 0.1 and 10. This is very encouraging for our tool whose efficiency is still in an upgradeable stage, compared to the tool of [2] that relies on decades of implementation efforts for the LLVM optimizer and the Boogie verifier.

5 Related Work

This paper deals with static program verification for **information-flow tracking** [34]. Different formal techniques have been used in this area. The type-based approach [29] provides an elegant modular verification approach but requires program annotations, especially for each program function. Because a same function can be called in very different security contexts, providing an expressive annotation language is challenging and annotating programs is a difficult task. This approach has been mainly proposed for programming language with strong type guarantees such as Java [29] or ML [31]. The deductive approach [14] is based

on more expressive logics than type systems and then allows to express subtle program invariants. On the other hand, the loop invariant annotation effort requires strong formal method expertise and is very much time consuming. The static analysis approach only requires minimal annotation (the input taints) and then tries to infer all the remaining invariants in the restricted analysis logic. This approach has been followed to track efficiently implicit flows using program dependence graphs [20,33]. We also follow a static approach but our backbone technique is an advanced value analysis for C, that we use to infer fine-grained memory separation properties and finely track taints in an unfolded call graph of the program. Building a program dependence graph for memory is a well known challenge and scaling this approach to a Verasco (or Astrée) memory analysis is left for further work.

This paper deals however with a restricted notion of information flow: **constant-time security**. Here, implicit flow tracking is simplified since we must reject[1] control-flow branching that depends on secret inputs. Our abstract interpretation approach proposes to accompany a taint analysis with a powerful value analysis. The tool tis-ct [35] uses a similar approach but based on the Frama-C value analysis, instead of Verasco (and its Astrée architecture). The tool is developed by the TrustInSoft company and not associated with any scientific publication. It has been used to analyze OpenSSL. Frama-C and Verasco value analysis are based on different abstract interpretation techniques and thus the tainting power of tis-ct and our tool will differ. As an example of difference, Verasco provides relational abstraction of memory contents while tis-ct is restricted to non-relational analysis (like intervals). CacheAudit [17] is a also based on abstract interpretation but analyze cache leakage at binary level. Analysing program at this low level tempers the inference capabilities for memory separation, because the memory is seen as a single memory block. Verasco benefits from a source level view where each function has its own region for managing local variables.

In a previous work of the second author [5], C programs where compiled by CompCert to an abstraction of assembly before being analyzed. A simple data-flow analysis was then performed, flow insensitive for every memory block except the memory stack, and constant-time security was verified. The precision of this approach requires to fully inline the program before analysis. It means that every function call was replaced by its function body until no more function call remained. This has serious impact on the efficiency of the analysis and a program like `curve25519-donna` was out of reach. The treatment of memory stack was also very limited since no value analysis was available at this level or program representation. There was no way to finely taint an array content if this array laid in the stack (which occurs when C arrays are declared as local variables). Hence, numerous manual program rewritings were required before analysis. Our current approach releases these restrictions but requires more trust on the compiler (see our discussion in the conclusion).

[1] We could accept some of them if we were able to prove that all branches provide a similar timing behavior.

A very complete treatment of constant-time security has been recently proposed by the ct-verif tool [2]. Its verification is based on a reduction of constant time security of a program P to safety of a *product program* Q that simulates two parallel executions of P. The tool is based on the LLVM compiler and operates at the LLVM bytecode level, after LLVM optimizations and alias analyses. Once obtained, the bytecode program is transformed into a product program which, in turn, is verified by the Boogie verifier [4] and its traditional SMT tool suite. In Sect. 4, we made a direct experimental comparison with this tool. We list here the main design differences between this work and ours. First we do not perform the analysis at a similar program representation. LLVM bytecode is interesting because one can develop analyses that benefit from the rich collection of tools provided by the LLVM platform. For example, [2] benefits from LLVM data-structure analysis [24] to partition memory objects into disjoint regions. Still, compiler alias analyses are voluntarily limited because compilers translate programs in almost linear time. Verasco (and its ancestor Astrée) follows a more ambitious approach and tracks very finely the content of the memory. Using Verasco requires a different tool design but opens the door for more verified programs, as for example the SHA-256 example. Second, we target a more restricted notion of constant-time security than [2] which relaxes the property with a so-called notion of *publicly observable outputs*. The extension is out of scope of our current approach but seems promising for specific programs. Only one program in our current experiment is affected by this limitation. At last, we embed our tool in a more foundational semantic framework. Verasco and CompCert are formally verified. It leaves the door open for a fully verified constant-time analyzer, while a fully verified ct-verif tool would require to prove SMT solvers, Boogie verifier and LLVM. The Vellvm [37] is a first attempt in the direction of verifying the LLVM platform, but it is currently restricted to a core subset (essentially the SSA generation) of the LLVM passes, and suffers from time-performance limitations.

Other approaches rely on dynamic analysis (e.g. [13] that extends of Valgrind in order to check constant-address security) or on statistical analysis of execution timing [32]. These approaches are not sound.

6 Conclusion

In this paper, we presented a methodology to ensure that a software implementation is constant time. Our methodology is based on advanced abstract interpretation techniques and scales on commonly used cryptographic libraries. Our implementation sits in a rich foundational semantic framework, Verasco and CompCert, which give strong semantic guarantees. The analysis is performed at source level and can hence give useful feedback to the programmer that needs to understand why his program is not constant time.

There are two main directions for future work. The first one concerns semantic soundness. By inspecting CompCert transformation passes, we conjecture that they preserve the constant-time property of source programs we successfully analyze. We left as further work a formal proof of this conjecture. The second

direction concerns expressiveness. In order to verify more relaxed properties, we could try to mix the program-product approach of [2] with the Verasco analysis. The current loop invariant inference and analysis of [2] are rather restricted. Using advanced alias analysis and relational numeric analysis could strengthen the program-product approach, if it was performed at the same representation level as Verasco.

References

1. Aciiçmez, O., Koç, Ç.K., Seifert, J.-P.: On the power of simple branch prediction analysis. In: ACM Symposium on Information, Computer and Communications Security (2007)
2. Almeida, J.B., et al.: Verifying constant-time implementations. In: 25th USENIX Security Symposium, USENIX Security 16 (2016)
3. Andrysco, M., et al.: On subnormal floating point and abnormal timing. In: Proceedings of the 2015 IEEE Symposium on Security and Privacy (2015)
4. Barnett, M., et al.: Boogie: a modular reusable verifier for object-oriented programs. In: Proceedings of FMCO 2005 (2005)
5. Barthe, G., et al.: System-level non-interference for constant-time cryptography. In: Conference on Computer and Communications Security (CCS) (2014)
6. Bernstein, D.J.: Curve25519: new Diffie-Hellman speed records. In: Public Key Cryptography - PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography (2006)
7. Bernstein, D.J., Lange, T., Schwabe, P.: The security impact of a new cryptographic library. In: International Conference on Cryptology and Information Security in Latin America (2012)
8. Blanchet, B., et al.: A static analyzer for large safety-critical software. In: PLDI (2003)
9. Blazy, S., Laporte, V., Pichardie, D.: An abstract memory functor for verified C static analyzers. In: International Conference on Functional Programming (ICFP 2016) (2016)
10. Bos, J.W., et al.: Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In: IEEE Symposium on Security and Privacy, SP 2015 (2015)
11. Canvel, B., Hiltgen, A., Vaudenay, S., Vuagnoux, M.: Password interception in a SSL/TLS channel. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 583–599. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45146-4_34
12. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Symposium on Principles of Programming Languages, POPL 1977 (1977)
13. ctgrind. https://github.com/agl/ctgrind
14. Darvas, Á., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Proceedings of 2nd International Conference on Security in Pervasive Computing (2005)
15. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**, 236–243 (1976)
16. donna. https://code.google.com/archive/p/curve25519-donna
17. Doychev, G., et al.: CacheAudit: a tool for the static analysis of cache side channels. In: USENIX Conference on Security (2013)

18. Al Fardan, N.J., Paterson, K.G.: Lucky thirteen: breaking the TLS and DTLS record protocols. In: Symposium on Security and Privacy (SP 2013) (2013)
19. Feret, J.: Static analysis of digital filters. In: European Symposium on Programming (ESOP 2004) (2004)
20. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. Int. J. Inf. Sec. **8**, 399–422 (2009)
21. Hedin, D., Sabelfeld, A.: A perspective on information-flow control. In: Software Safety and Security - Tools for Analysis and Verification (2012)
22. Jourdan, J.-H., et al.: A formally-verified C static analyzer. In: Symposium on Principles of Programming Languages, POPL 2015 (2015)
23. Kocher, P.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Advances in Cryptology - CRYPTO 1996 (1996)
24. Lattner, C., Lenharth, A., Adve, V.S.: Making contextsensitive points-to analysis with heap cloning practical for the real world. In: Conference on Programming Language Design and Implementation, PLDI 2007 (2007)
25. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**, 107–115 (2009)
26. mbed TLS (formerly known as PolarSSL). https://tls.mbed.org/
27. Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2006) (2006)
28. Miné, A.: The octagon abstract domain. In: Higher-Order and Symbolic Computation (2006)
29. Myers, A.C.: JFlow: practical mostly-static information flow control. In: Symposium on Principles of Programming Languages, POPL 1999 (1999)
30. Open Quantum Safe. https://openquantumsafe.org/
31. Pottier, F., Simonet, V.: Information flow inference for ML. ACM Trans. Program. Lang. Syst. **25**, 117–158 (2003)
32. Reparaz, O., Balasch, J., Verbauwhede, I.: Dude, is my code constant time. In: Proceedings of DATE 2017 (2017)
33. Rodrigues, B., Quintão Pereira, F.M., Aranha, D.F.: Sparse representation of implicit flows with applications to side-channel detection. In: Compiler Construction (2016)
34. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**, 5–19 (2003)
35. TIS-CT. http://trust-in-soft.com/tis-ct/
36. Wheeler, D.J., Needham, R.M.: TEA, a tiny encryption algorithm. In: Fast Software Encryption: Second International Workshop Leuven (1995)
37. Zhao, J.: et al.: Formalizing the LLVM intermediate representation for verified program transformation. In: Symposium on Principles of Programming Languages, POPL 2012 (2012)