Review: Verifying Constant-Time Implementations

Mahyar Emami, Rishabh Iyer, Sahand Kashani
firstname.lastname@epfl.ch

January 11, 2022

1 Introduction

Timing attacks—attacks that extract secrets by measuring timing differences under adversary-controlled inputs—on cryptographic libraries [9, 12] pose a major challenge to information security today. Popular cryptographic libraries (e.g., OpenSSL [7]) run on millions of devices; hence a vulnerability in such a library has the potential to compromise all such devices simultaneously.

Constant-Time programming is the most effective software countermeasure against such attacks. Constant-time programming involves rewriting a program such that (1) its control flow does not depend on program secrets, (2) it does not perform any secret-dependent memory accesses, and (3) it does not use variable-latency instructions to operate on secrets.

However, writing and/or reasoning about constant-time programs is challenging and prone to errors since it typically involves the use of low-level programming languages and programming practices that deviate from software engineering principles. For example, two constant-time violations[1] were found in Amazon's s2n library [1] soon after its release with the second exploiting a timing-related vulnerability introduced when fixing the first.

In this work, Almeida et. al [8] propose using formal methods to *automatically* verify whether a program runs in constant time. To do so, they provide a precise framework to model constant-time properties(§2), a sound and complete reduction of constant-timeliness of input programs to assertion safety of a self-product(§3) and design, evaluate an automated tool that verifies the constant-timeliness of cryptographic algorithms from widely used libraries within seconds(§4).

2 Formalization of Constant Time

Here we summarize the authors' formalization of constant-timeliness for a simple, high-level, structured programming language. Fig. 1 lists the lan-

[1] See pull requests #147 and #179 in [1]

guage's syntax. The operational semantics for the language's primitives are standard and are listed in Fig. 2 for reference.

$$
\begin{aligned}
p ::= {} & \texttt{skip} \mid x\texttt{[}e_1\texttt{]} \texttt{ := } e_2 \mid \texttt{assert } e \mid \texttt{assume } e \mid p_1\texttt{; } p_2 \\
& \mid \texttt{if } e \texttt{ then } p_1 \texttt{ else } p_2 \mid \texttt{while } e \texttt{ do } p
\end{aligned}
$$

Figure 1: Syntax of language used for formalization

$$
\frac{s' = s[\langle x, s(e_1)\rangle \mapsto s(e_2)]}{\langle s,\, x\texttt{[}e_1\texttt{] := } e_2\rangle \to \langle s', \texttt{skip}\rangle}
\qquad
\frac{s' = s \text{ if } s(e) \text{ else } \bot}{\langle s,\, \texttt{assert } e\rangle \to \langle s', \texttt{skip}\rangle}
$$

$$
\frac{s(e) = \texttt{true}}{\langle s,\, \texttt{assume } e\rangle \to \langle s, \texttt{skip}\rangle}
\qquad
\frac{\langle s, p_1\rangle \to \langle s', p_1'\rangle}{\langle s,\, p_1\texttt{; } p_2\rangle \to \langle s',\, p_1'\texttt{; } p_2\rangle}
$$

$$
\frac{}{\langle s,\, \texttt{skip; } p\rangle \to \langle s, p\rangle}
\qquad
\frac{i = 1 \text{ if } s(e) \text{ else } 2}{\langle s,\, \texttt{if } e \texttt{ then } p_1 \texttt{ else } p_2\rangle \to \langle s, p_i\rangle}
$$

$$
\frac{p' = (p\texttt{; while } e \texttt{ do } p) \text{ if } s(e) \text{ else } \texttt{skip}}{\langle s,\, \texttt{while } e \texttt{ do } p\rangle \to \langle s,\, p'\rangle}
$$

Figure 2: Operational semantics of language in Fig. 1

Modeling Program Execution: A state s maps variables x and indices $i \in$ to values $s(x, i)$ and we use $s(e)$ to denote the value of expression e in state s. The distinguished error state $\bot$ represents a state from which no transition is enabled. A *configuration* $c = \langle s, p\rangle$ is a state s along with a program p to be executed, and an *execution* is a sequence $c_1, c_2, ...c_n$ of configurations such that $c_i \to c_{i+1}$ for $0 < i < n$. The execution is *safe* unless $c_n = \langle \bot, \_\rangle$; it is *complete* if $c_n = \langle \_, skip\rangle$; and it is an *execution of program p* if $c_1 = \langle \_, p\rangle$. A program p is *safe* if all of its executions are safe.

Given a set of X of program variables, two configurations $\langle s_1, \_\rangle$ and $\langle s_2, \_\rangle$ are *X-equivalent* when $\forall x \in X, i \in \mathbb{N} : s_1(x, i) = s_2(x, i)$. Executions $c_1...c_2$ and $c_1'...c_n'$ are initially *X-equivalent* when c_1 and c_1' are *X-equivalent* and finally *X-equivalent* when c_n and c_n' are *X-equivalent.*

Modeling Leakages: A *leakage model* L maps program configurations c to *observations* $L(c)$, and extends to executions, mapping $c_1, c_2, ...c_n$ to the *observation* $L(c_1, c_2, ...c_n) = L(c_1) \cdot L(c_2) \cdots L(c_n)$. Two executions α and β are *indistinguishable* when $L(\alpha) = L(\beta)$.

Definition of Constant-Time Security: *A program p with a set X_i of inputs declared publicly observable and a set X_o of outputs declared publicly observable is constant-time secure when all of its initially X_i-equivalent and finally X_o-equivalent executions are executable.*

Sources of leakage considered: The authors consider three leakage models in this work—(1) branch-based leakage, (2) memory access-based leakage, and (3) leakage based on variable-latency instructions.

Branch-based leakages expose the valuations of branch conditions. For instance, here are the leakage models for branches in the example high-level language— (1) $\langle$ s, if e then p_1 else $p_2\rangle \mapsto s(e)$, (2) $\langle$ s, while e do p$\rangle \mapsto s(e)$

Memory access-based leakages expose the addresses accessed in load and store instructions. In the simple language of while programs, this is equivalent to exposing the indexes to program variables read from and written to at each statement.

Finally, the time required to execute an instruction may vary based on the provided operands on modern processors. For instance, integer division on x86 processors depends on the size of the two operands [5]). To account for such leakages, the authors add leakage models for all such instructions. Here, we only show the model for division - $\langle$ s, x $= e_1/e_2\rangle \mapsto \mathit{sizeof}(e_1, e_2)$

3 Reducing security to safety

We now illustrate (using an example) how the authors use the leakage rules in §2 to reduce constant-time security to assertion safety.

We use the program listed in Fig. 3 as our running example. This program copies a sub-array of length `sub_len`, starting at index `l_idx`, from array `in` to array `out`. We assume that the starting addresses and lengths of both arrays are publicly observable but the value of `l_idx` and the array contents must be kept a secret. Note, this program is *not constant-time secure* since the branches on line 5 clearly leak information about `l_idx`.

```
1 void copy_subarray(uint8 *out, const uint8 *in,
2     uint32 len, uint32 l_idx, uint32 sub_len){
3   uint32 i,j;
4   for(i=0,j=0; i<len; i++){
5     if((i >= l_idx) && (i<l_idx + sub_len)){
6       out[j] = in[i];
7       j++;
8     }
9   }
10 }
```

Figure 3: Running example - sub-array copy

The authors reduce the constant-time security of the original program (P) to the assertion safety of a second program (Q) that is the *self-product* of P. Said differently, Q is a program in which two abstract executions of P take place simultaneously, with the two executions only differing in the value of *secret* inputs and outputs. Fig. 4 .a shows how such a self-product can be constructed ($\hat{p}$ is the program p with all variables renamed). First, each public input is assumed to be equal in both executions, then a guard and instrumentation are recursively applied to each subprogram. The guards assert the equality of leakage functions for each subprogram p and its variable-renaming $\hat{p}$. The essence of the transformation lies in the instrumentation (Fig. 4 .b) which reduces constant-time security to assertion safety. The authors formalize this reduction from constant-time security to safety in Coq and prove that it is sound for all safe input programs (i.e., a security verdict is always correct) and complete for programs where information about public outputs is not taken into consideration in the security analysis (i.e., an insecurity verdict is always correct).

$\mathsf{product}(p)$	`assume` $x{=}\hat{x}$ `for` $x \in X_{\mathsf{i}}$`;` $\mathsf{together}(p)$
$\mathsf{together}(p)$	$\mathsf{guard}(p)$`;` $\mathsf{instrument}[\lambda p.(p\,;\hat{p}),\mathsf{together}](p)$
$\mathsf{guard}(p)$	`assert` $L(p){=}L(\hat{p})$

(a) Program product construction rules

_	$\mathsf{instrument}[\alpha,\beta](\_)$
`skip`	`skip`
x`[`e_1`] :=` e_2	$\alpha(x$`[`e_1`] :=` $e_2)$
`assert` e	`assert` e
`assume` e	`assume` e
p_1`;` p_2	$\beta(p_1)$`;` $\beta(p_2)$
`if` e `then` p_1 `else` p_2	`if` e `then` $\beta(p_1)$ `else` $\beta(p_2)$
`while` e `do` p	`while` e `do` $\beta(p)$

(b) Instrumentation rules.

Figure 4: Program product

Fig. 5 illustrates the self-product for our running example. The resultant program is instrumented with assertions to ensure leakage remains the same at every step of execution. Since `l_idx` is a private input to the program, its renaming cannot be assumed equal. Consequently, the assertion on line 8 fails, demonstrating that assertions in the self-product do not hold if the given program does not run in constant time.

```
1  assume in = in̂;
2  assume out = oût;
3  assume len = lên;
4  assume sub_len = sub_̂len;
5  i = 0; î = 0; j = 0; ĵ = 0;
6  assert (i < len) = (î < lên); // trivial
7  while (i < len) do:
8      assert ((i ≥ l_idx) && (i < l_idx + sub_len))
9          = ((î ≥ l_îdx) && (î ≥ l_îdx + sub_̂len) // fails;
10     if ((i ≥ l_idx) && (i < l_idx + sub_len)) then
11         assert i = î && j = ĵ; // trivial
12         out[j] = in[i]; oût[ĵ] = in̂[î];
13         j = j + 1; ĵ = ĵ + 1;
14     i = i + 1; î = î + 1;
```

Figure 5: Example program product of the sub-array copy program.

4 Evaluation

The authors implement the algorithm outlined in §3 in a publicly available tool called `ct-verif` [3]. `ct-verif` takes as input the LLVM implementation of a cryptographic algorithm and outputs either proof of constant-timeliness or a counter-example showing how the property is violated. Under the covers, `ct-verif` leverages the SMACK verification tool [16] to translate the LLVM to Boogie [2] code. It then performs its reduction on the Boogie code and applies the Boogie verifier to the resulting program.

The authors evaluate their tool on examples from widely used cryptographic libraries including OpenSSL [7], NaCl [6], FourQlib [11] and `curve25519-donna` [4]. The examples include encryption algorithms, hash functions, fixed-point arithmetic and elliptic-curve arithmetic. The size of the examples ranges from $50 - 1200$ lines of C code.

The authors demonstrate that `ct-verif` is typically able to verify that the examples are constant-time secure within seconds—most functions require $\leq 30s$ while the largest function requires $273s$. These results show that `ct-verif` can be easily integrated into the day-to-day development of cryptographic algorithms.

5 Takeaways

This work formalizes the previously imprecise notion of constant-time programming, in addition to providing a publicly available and fully automated tool that can verify (in seconds) whether implementations of cryptographic algorithms from off-the-shelf libraries adhere to the notion of constant-time. This is, undoubtedly, an impressive result.

That said, the paper does have a few weaknesses:

Firstly, `ct-verif` requires LLVM implementations which is problematic for two reasons. Cryptographic code is often written directly in assembly due to the performance benefits. For instance, Vale [10] demonstrates that OpenSSL assembly implementations outperform their C counterparts by up to 50%. Further, verifying that LLVM code runs in constant time does not guarantee that the executable runs in constant time [13]. To the authors' credit, they do discuss such scenarios.

Secondly, the formalization of constant-time as presented by the papers is significantly weakened by the advent of micro-architectural attacks ([15, 14]). To be fair to the authors, such attacks became popular after their paper was published. Further, one could argue that such attacks are not a property of the implementation, and are a menace that the underlying hardware/operating system must tackle.

Finally, the authors do not present a single example where their "automated" approach fails to verify constant-timeliness in reasonable time. As such, the results look a little too good to be true.

During the course of this project, we plan to investigate the first and third aspects of `ct-verif` more thoroughly. Specifically, we aim to identify where the tool falls short and also patch it with LLVM-passes that can preserve constant timeliness.

References

[1] Amazon s2n-tls Github Repository. `https://github.com/aws/s2n-tls`.

[2] Boogie Github Repository. `https://github.com/boogie-org/boogie`.

[3] CT-Verif Github Repository. `https://github.com/imdea-software/verifying-constant-time`.

[4] curve25519-donna. `https://code.google.com/archive/p/curve25519-donna/`.

[5] Intel 64 and 32 bit architecture manuals. `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`.

[6] Networking and Cryptographic Library (NaCl). `https://nacl.cr.yp.to/`.

[7] OpenSSL Github Repository. `https://github.com/openssl/openssl`.

[8] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security Symposium*, 2016.

[9] Daniel J. Bernstein. Cache-timing attacks on aes. Technical report, 2005.

[10] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *USENIX Security Symposium*, 2017.

[11] Craig Costello and Patrick Longa. Fourq: four-dimensional decompositions on a q-curve over the mersenne prime. In *Advances in Cryptology - ASIACRYPT, 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015*, 2015.

[12] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. Make sure dsa signing exponentiations really are constant-time. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

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

[14] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: exploiting speculative execution. *Commun. ACM*, 2020.

[15] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, 2018.

[16] Zvonimir Rakamaric and Michael Emmi. SMACK: decoupling source language details from verifier implementations. In *Computer Aided Verification CAV*, 2014.