Enforcing fine-grained constant-time policies

Basavesh Ammanaghatta Shivakumar
MPI-SP
Bochum, Germany
basavesh.shivakumar@mpi-sp.org

Gilles Barthe
MPI-SP
Bochum, Germany
IMDEA Software Institute
Pozuelo de Alarcón, Spain
gbarthe@mpi-sp.org

Benjamin Grégoire
Université Côte d'Azur, Inria
Sophia Antipolis, France
benjamin.gregoire@inria.fr

Vincent Laporte
Université de Lorraine, CNRS, Inria, LORIA
F-54000 Nancy, France
vincent.laporte@inria.fr

Swarn Priya
Université Côte d'Azur, Inria
Sophia Antipolis, France
swarn.priya@inria.fr

ABSTRACT

Cryptographic constant-time (CT) is a popular programming discipline used by cryptographic libraries to protect themselves against timing attacks. The CT discipline aims to enforce that program execution does not leak secrets, where leakage is defined by a formal leakage model. In practice, different leakage models coexist, sometimes even within a single library, both to reflect different architectures and to accommodate different security-efficiency trade-offs.

Constant-timeness is popular and can be checked automatically by many tools. However, most sound tools are focused on a baseline (BL) leakage model. In contrast, (sound) verification methods for other leakage models are less developed, in part because these models require modular arithmetic reasoning. In this paper, we develop a systematic, sound, approach for enforcing fine-grained constant-time policies beyond the BL model. Our approach combines two main ingredients: a verification infrastructure, which proves that source programs are constant-time, and a compiler infrastructure, which provably preserves constant-timeness for these fine-grained policies. By making these infrastructures parametric in the leakage model, we achieve the first approach that supports fine-grained constant-time policies. We implement the approach in the Jasmin framework for high-assurance cryptography, and we evaluate our approach with examples from the literature: OpenSSL and wolfSSL. We found a bug in OpenSSL and provided a formally verified fix.

KEYWORDS

Secure Compilation, Cryptographic Constant-Time

1 INTRODUCTION

Timing attacks [27] are a class of side-channel attacks in which attackers monitor (and analyze) program execution time to learn about secret values used by these programs. Timing attacks remain an important concern for cryptographic libraries more than twenty-five years after their discovery. A pragmatic approach to minimize these attacks is to ensure that program leakage does not depend on secrets, using an *idealized* model of leakage. Many cryptographic libraries adopt this approach under the generic umbrella of *constant-time* cryptography. Over the last years, constant-time cryptography has also become a main target for verification [6, 26] and secure compilation [9, 14, 15].

Program Counter (PC)	Conditionals leak their guards
Baseline (BL)	PC + Memory R/W leak addresses
Cache line (CL)	PC + Mem. accesses leak cache lines
Time-Variable (TV)	BL + TV arithmetic operators leak
TV + CL	TV arithmetic operators leak + CL

Figure 1: Common leakage models

However, constant-time cryptography is based on a family of leakage models rather than a single model; see Figure 1 for a short description of some key models. In general, these models differ subtly based on the considered threat model, the intended target platform, and the tractability of the constant-time verification problem for this model. In practice, cryptographic libraries such as OpenSSL optimize their implementations for each leakage model and therefore provide one implementation per leakage model. Unfortunately, the multiplicity of implementations and leakage models can lead to a false sense of security. We provide two potential scenarios below:

- a library is verified for constant-time, but only for a specific leakage model, so only the functions relevant to this model are checked. For instance, OpenSSL provides multiple implementations of the same crypto routines optimized for different leakage models. Therefore, when a library such as OpenSSL is verified for constant-time, it is likely that only the functions relevant to the BL leakage models are checked. In contrast, no guarantee is given for functions that target the CL model, which is harder to verify;
- a library is verified for constant-time in a weaker leakage model than intended, because of inherent limitations in the verification technology, and as a consequence, it may still leak in the intended leakage model. For instance, some crypto routines in (earlier versions of) OpenSSL are provably secure in the BL leakage model but are insecure in the TV model and vulnerable to practical timing attacks—such as Lucky13 [1].

These two scenarios reflect the existence of a potentially dangerous gap in computer-aided cryptography. This paper considers the broad problem of generalizing existing works on verification and secure compilation of the constant-time policy to cover different leakage models.

Contributions

The first contribution, which motivates for this work, is a review of implementations of MEE-CBC, a component of the TLS protocol, in the cache line (CL) leakage model defined as follows. In the CL model, in contrast to the BL model where memory accesses leak the exact address, memory accesses leak the cache line of the address accessed, i.e. the address divided by the size of the cache line. This makes constant-time analysis challenging and error-prone because modular arithmetic reasoning is required for proving the equality of leakages. In fact, we show that the OpenSSL implementation of MEE-CBC tailored to the CL leakage model violates its intended constant-time policy. We make no attempt to exploit this leakage in a timing attack, mainly because a cheap fix closes the leakage.

The second and main technical contribution is a mechanized proof in the Coq proof assistant that the Jasmin compiler preserves a class of fine-grained constant-time policies. This class encompasses the BL policy as well as policies that capture time-variable instructions and fine-grained leakage models of memory accesses. The proof takes the form of an instrumented correctness theorem [15] and shows that leakage of assembly programs can be computed deterministically from leakage of source programs.

Our third contribution is a set of formal proofs that previously unverified cryptographic code is constant-time in a (non-baseline) leakage model. This includes Langley's patch to Lucky13 and our own fix of OpenSSL. The proofs are carried via an embedding of Jasmin source code into EasyCrypt and using EasyCrypt's implementation of relational Hoare logic. Although the proofs are carried for Jasmin programs, our certified compiler carries the guarantees to the generated code.

Overall, our work allows expands tool support for the constant-time policy, which has been very beneficial and has contributed to making cryptographic libraries more robust against timing attacks, to a rich set of policies.

Supplementary material. The full development is provided as supplementary material at https://github.com/jasmin-lang/jasmin/tree/constant-time-op

2 BACKGROUND AND EXAMPLE

This section gives a background on Jasmin and EasyCrypt, explains the different leakage models, and presents our motivating example.

2.1 Background on Jasmin

Jasmin is a framework for high-speed and high-assurance cryptography [2, 3]. The Jasmin language smoothly combines high-level and low-level constructs so as to support "assembly in the head" programming. Programmers can control many low-level details that are performance-critical: instruction selection and scheduling, what registers to spill and when etc. They can also rely on high-level abstractions (variables, functions, arrays, loops, etc.) to structure their code and make it more amenable to formal verification. Indeed, the Jasmin infrastructure has been used to implement cryptographic constructions that are as efficient as state-of-the-art handwritten assembly and, nonetheless, formally verified for correctness and security with machine-checked proofs. These implementations notably include scalar multiplication on the Curve25519 elliptic curve [2], the ChaCha20 stream cipher [3], and the SHA-3 hash function [4]. Jasmin programs are compiled to assembly code (currently targeting the x86_64 architecture) using a verified compiler formally proved using Coq.

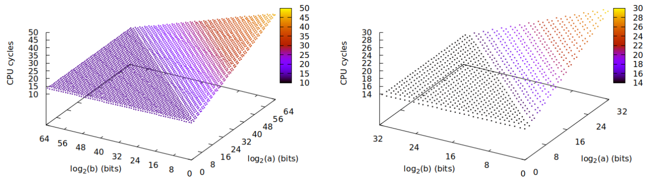

Figure 2: Timing behavior of the div instruction on a x86 microprocessor (AMD EPYC 7F52) with 64-bit (left) and 32-bit (right) operands. It computes at once both quotient and modulo of its arguments a and b. Different microprocessor models exhibit different timing profiles.

2.2 Constant-time leakage models

Baseline (BL) leakage model. This is the simplest leakage model but not weakest. This model assumes:

- branching statements leak values of their guards;
- memory operations leak the addresses accessed;
- nothing else leaks.

This model is the basis of the baseline constant-time policy, which mandates that leakage does not depend on secrets. This policy is appealing for three main reasons. First, it captures many timing attacks from the literature. Second, the model is quite tractable and can be used effectively by cryptographic engineers to guide their implementations. Moreover, there is a large spectrum of automated tools [6] for (dis)proving that programs satisfy the baseline constant-time property, i.e. their leakage is independent of secrets. Third, there is a recent line of work [9, 15] that establishes the preservation of the baseline constant-time policy for some realistic compilers.

Time-variable (TV) leakage model. The baseline leakage model does not capture leakage resulting from time-variable instructions. Such instructions, which leak information about their operands, are pervasive in all modern architectures; for the x86 architecture, they include division and modulo (see Figure 2). As a consequence, constant-time programs may still leak through their time-variable instructions. In specific circumstances, this leakage may be exploited to recover cryptographic keys. To address this issue, the TV leakage model strengthens the BL leakage model by making time-variable arithmetic instructions leak a function of their operands. In this paper, we assume that the modulo operation leaks the base-2 integer logarithm of its operands.

Cache line (CL) leakage model. The baseline constant-time leakage model assumes that memory operations leak the addresses of the memory accessed. This assumption helps in protecting against attacks that exploit cache-bank conflicts but also makes programs harder to write and less efficient. As a consequence, cryptographic libraries often provide implementations for the CL leakage model. In this leakage model, memory accesses leak the cache line of the addresses being accessed. In this paper, memory accesses leak the address divided by the length of the cache line (32, 64, ...).

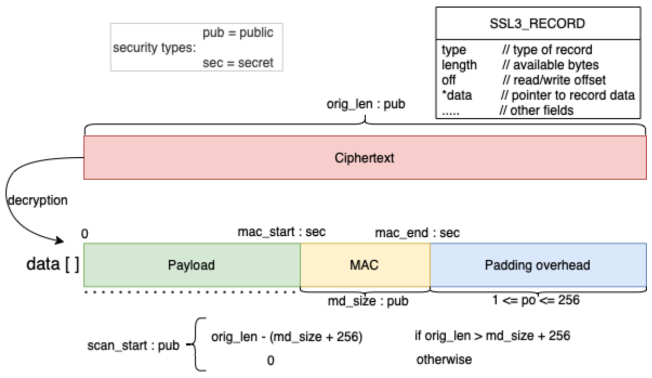

Figure 3: Memory layout of a decrypted SSL3 record

Summary of models.

- the baseline (BL) leakage model, where guards and memory addresses are leaked;
- the time-variable (TV) leakage model, where guards and memory addresses are leaked, and time-variable arithmetic instructions leak a function of their operands;
- the cache line model (CL), where guards and cache lines of memory addresses are leaked;
- the TV + CL model, combining the TV and CL models.

2.3 Background on EasyCrypt

EasyCrypt [10] is a proof assistant used for reasoning about cryptographic primitives. EasyCrypt features an ambient higher-order logic with a backend to SMT solvers as well as several program logics to reason about (probabilistic) imperative programs written in a core language. EasyCrypt was originally used for proving security properties for several cryptographic primitives [7, 12, 13]. More recently, it has been used to prove functional correctness and side-channel security of cryptographic primitives written in Jasmin. For the latter, proofs proceed by embedding Jasmin programs with their leakage into EasyCrypt, and by using relational Hoare logic to prove that leakage is independent of secrets. In prior work, focusing on the BL model, application of relational Hoare logic is fully automated. We provide more detail in Section 5.

2.4 Motivating Example: MEE-CBC

MEE-CBC (MAC-then-Encode-then-CBC-Encrypt) is an authenticated encryption scheme used in the TLS 1.2 ciphersuite. In 2013, AlFardan and Paterson [1] designed Lucky13, a sophisticated timing attack against several open-source cryptographic libraries supporting MEE-CBC. In response to the attack, several libraries developed new implementations of MEE-CBC that loosely follow the constant-time programming discipline. However, proving that these implementations are constant-time according to the baseline leakage model is not always possible, as some implementations use time-variable instructions, or optimize code in a way that degrades security to a weaker model. In this section, we review different implementations of two core functions of MEE-CBC and analyze their security in the four different leakage models discussed in the previous section.

```
/* public: md_size, scan_start */
/* secret: mac_start */
fn rotate_offset_BL(reg u32 md_size, mac_start, scan_start) ⟶ reg u32 {
    reg u32 rotate_offset;
    rotate_offset = mac_start;
    rotate_offset -= scan_start;
    rotate_offset = rotate_offset % md_size;
    return rotate_offset;
}
```

```
/* pre: 16 ≤ md_size ≤ 64 ∧ 0 ≤ mac_start - scan_start < 256 */
/* public: md_size, scan_start */
/* secret: mac_start */
fn rotate_offset_TV(reg u32 md_size, mac_start, scan_start) ⟶ reg u32 {
    reg u32 div_spoiler;
    reg u32 rotate_offset;
    div_spoiler = md_size;
    div_spoiler <<= 23;
    rotate_offset = mac_start;
    rotate_offset -= scan_start;
    rotate_offset += div_spoiler;
    rotate_offset = rotate_offset % md_size;
    return rotate_offset;
}
```

Figure 4: Two implementations computing the rotation offset

CBC decoding. We briefly review the CBC decoding function, in which the functions we study are used. Figure 3 provides a pictorial description of the inputs to the function ssl3_cbc_copy_mac. This function takes four arguments, 1. rec pointer to a ssl3 record structure which contains a pointer to data buffer and a secret length field which denotes the length after removing padding overhead; 2. out buffer, where extracted message authentication code (MAC) will be copied to; 3. md_size denotes the MAC size and 4. orig_len representing the record's original length, including padding.

Initially, ciphertext is decrypted into data buffer, containing the payload, the MAC and finally the padding overhead. The goal is to copy the MAC in constant-time to the out buffer. Unfortunately, we cannot simply copy the MAC, because mac_start and mac_end are secret. We also do not want to scan the whole data buffer, for efficiency reasons.

However, we know that orig_len is public, and that the padding overhead is at most 256 bytes long. Therefore, it suffices to begin copying from $\text{scan\_start} = \max(0, \text{orig\_len} - (\text{md\_size} + 256))$. However, there is a twist: constant-time copying might yield byte-wise rotated values. Therefore, in order to recover the original MAC, one must compute the offset and perform the rotation relative to this offset in constant-time.

Computing the rotation offset. Figure 4 shows the functions rotate_offset_BL and rotate_offset_TV for computing the offset. For readability, we annotate these functions with security levels and preconditions.

The first function rotate_offset_BL is a simple arithmetic program that computes $(\text{mac\_start} - \text{scan\_start}) \bmod \text{md\_size}$. This function is trivially constant-time in the baseline (BL) constant-time leakage model, as it does not branch nor perform memory accesses. However, the variable mac_start is secret; hence this function is not constant-time in the time-variable (TV) leakage model.

The second function rotate_offset_TV is a Jasmin implementation of Langley's original fix to Lucky13 and is constant-time

```
 1 /* public: md_size, out */
 2 /* secret: rotate_offset */
 3 fn rotate_mac_BL (reg u32 md_size, rotate_offset,
 4                   reg u64 out, stack u8[128] rotated_mac) {
 5   reg u64 i, j;
 6   reg u32 old, new, zero, ro;
 7   zero = 0;
 8   // ro = (-rotate_offset) % md_size
 9   ro = opp_mod(rotate_offset, md_size);
10   i = 0;
11   while (i < md_size) {
12     j = 0;
13     while (j < md_size) {
14       old = (32u) (u8)[out + j];
15       new = (32u) rotated_mac[(int) i];
16       new = old if j != ro;
17       (u8)[out + j] = new;
18       j += 1;
19     }
20     ro += 1; ro = zero if md_size <= ro;
21     i += 1;
22   }
23 }
```

```
 1 /* rotated_mac % 64 = 0 ∧ 0 ≤ rotate_offset ≤ mdsize ≤ 64*/
 2 /* public: md_size, out, rotated_mac */
 3 /* secret: rotate_offset */
 4 fn rotate_mac_CL (reg u32 md_size, rotate_offset,
 5                   reg u64 out, rotated_mac) {
 6   reg u8 new;
 7   reg u64 i, zero, ro;
 8   zero = 0;
 9   ro = (64u) rotate_offset;
10   i = 0;
11   while (i < md_size) {
12     new = (u8)[rotated_mac + ro];
13     (u8)[out + i] = new;
14     ro += 1; ro = zero if md_size <= ro;
15     i += 1;
16   }
17 }
```

Figure 5: Two implementations of MAC rotation

in the TV model. This is achieved by making the first argument large enough, by first setting $\mathsf{div\_spoiler} = \mathsf{md\_size} << 23$ and by setting rotate_offset to $\mathsf{div\_spoiler} + (\mathsf{mac\_start} - \mathsf{scan\_start})$. Note that the denominator (md_size) is a public value, so only the numerator (rotate_offset) needs to be patched in such a way. Under the assumptions on the parameters set by the precondition, we can prove that the change does not affect the result of the instruction and also makes leakage independent of mac_start. Indeed, the leakage of the modulo instruction is equal to:

$$\log_2\,(\mathsf{md\_size} \times 2^{23}),\ \log_2(\mathsf{md\_size})$$

and hence only depends on public values. To justify the above claim, note that by definition of the leakage model, the first component of the leakage is

$$\log_2(\mathsf{md\_size} \times 2^{23} + (\mathsf{mac\_start} - \mathsf{scan\_start}))$$
$$= \log_2(\mathsf{md\_size} \times 2^{23}).$$

The equality above follows from the precondition.

Rotating the MAC. Figure 5 presents the two Jasmin implementations. This code uses some Jasmin specific notation: the notation (u8)[p] is the Jasmin syntax for byte memory load/store at address p, the notation (64u)x is the syntax for zero-extension (i.e., cast from 32-bits to 64-bits).

```
#if defined(CBC_MAC_ROTATE_IN_PLACE)
j = 0;
for (i = 0; i < md_size; i++) {
  /* in case cache-line is 32 bytes, touch second line */
  ((volatile unsigned char *)rotated_mac)[rotate_offset ^ 32];
  out[j++] = rotated_mac[rotate_offset++];
  rotate_offset &= constant_time_lt_s(rotate_offset, md_size);
}
#else ...
```

Figure 6: Buggy C implementation of OpenSSL rotate_offset

The first implementation performs a nested loop. Before the loop ro is set to $(-\mathsf{rotate\_offset}) \bmod \mathsf{md\_size}$ (line 9). For each i, the inner loop writes to out buffer, if j equals rotate_offset, new value obtained at line 15 is written, else the old value is rewritten. The selection is done at line 16 using a conditional assignment, and will be compiled using a constant-time CMOVcc instruction. Line 20 computes $(\mathsf{rotate\_offset} + 1) \bmod \mathsf{md\_size}$ in constant-time.

This implementation is constant-time in the baseline leakage model, since branching statements (here the while loop) only depend on public data (i, j and md_size), and similarly for memory and array accesses, which only depend on i, j and out. The drawback is that the implementation performs a nested loop and so the copy is quadratic in md_size.

In contrast, the second implementation rotated_mac_CL performs a single loop. The code is straightforward. Observe that the leakage corresponding to branching instructions only depends on public data, concretely i and md_size. Next, we consider leakage from memory instructions. The instruction at line 13 only depends on public data (i and out), so it does not leak. However, the instruction at line 12 leaks the secret index rotate_offset at the first iteration. Hence the function is not constant-time in the BL leakage model. On the other hand, rotated_mac_CL is constant-time in the CL leakage model assuming that rotated_mac is 64-byte aligned memory pointer, and the MAC data will fit in a cache line—these assumptions correspond to the precondition. This is because in this model the instruction leaks: $\lfloor(\mathsf{rotated\_mac} + \mathsf{ro})/64\rfloor$. Since $\mathsf{rotated\_mac} \bmod 64 = 0$ and $0 \le \mathsf{ro} < \mathsf{md\_size} \le 64$, it follows:

$$\lfloor(\mathsf{rotated\_mac} + \mathsf{ro})/64\rfloor = \lfloor\mathsf{rotated\_mac}/64\rfloor.$$

Since the value of the pointer rotated_mac is public, the leakage does not depend on secrets.

OpenSSL bug and responsible disclosure. Figure 6 shows the code[1] used by OpenSSL in CL model to accommodate CPUs with 32-byte cache lines. Here rotated_mac is a 64-byte aligned buffer (i.e., its address is $64q$ for some q) and assumed that this data would fit into two 32-byte cache lines. The developer has added a dummy unoptimizable[2] first access to load rotated_mac[rotate_offset ^ 32] and then the actual load to ensure that they touch both lines in every iteration to make it look like constant-time. The comment

[1]still present in the master branch: https://github.com/openssl/openssl/blob/c9007bda79291179ed2df31b3dfd9f1311102847/ssl/record/tls_pad.c#L292

[2]Normally, a compiler can remove an unused load instruction since it does not change the semantic. However, volatile keyword prevents the compiler from performing this kind of optimization.

$e \in \mathsf{Expr} ::=$	x	variable
	$\mid c$	constant
	$\mid a[e]$	array read
	$\mid *e$	memory load
	$\mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$	conditional expression
	$\mid \mathsf{op}(e, \ldots, e)$	operator
$d \in \mathsf{Lval} ::=$	x	variable
	$\mid a[e]$	array write
	$\mid *e$	memory store
$i \in \mathsf{Instr} ::=$	$d := e$	assignment
	$\mid \mathsf{if}\ e\ \mathsf{then}\ i\ \mathsf{else}\ i$	conditional
	$\mid \mathsf{while}\ e\ \mathsf{do}\ i$	while loop
	$\mid \{i; \ldots; i\}$	sequencing

a ranges over array variables; x ranges over scalar variables

Figure 7: Syntax of Jasmin programs

in the code says it will touch the "second line". However, this is incorrect, and it touches "other line". According to 32-byte cache line model (CL_{32}), the address divided by 32 is leaked. Thus, for the two load operations, when $0 \leqslant \mathsf{rotate\_offset} < 32$, values $2q + 1$ and $2q$ are leaked (i.e second cache line is touched first and then the first line) and when $32 \leqslant \mathsf{rotate\_offset} < 64$, values $2q$ and $2q + 1$ are leaked (i.e first cache line is touched first and then the second). We have reported this issue and provided a formally verified fix (presented in Section 7.3.2) in the intended leakage model to OpenSSL developers. The bug is acknowledged and the fix is reviewed by OpenSSL developers.

Outline. In the next sections, we introduce a general setting to reason about fine-grained constant-time policies and prove that the above examples verify their policies using relational program logic. We also provide a generic proof that these policies are preserved by compilation. Finally, we implement our approach and evaluate its working on the illustrative and other examples.

3 FINE-GRAINED POLICIES IN JASMIN

This section introduces fine-grained leakage policies in the context of a core language inspired by Jasmin [2, 3].

3.1 Syntax & Semantics

Figure 7 introduces the syntax of our language[3]: it features scalar and array variables, memory accesses through pointer expressions, conditional expressions, a wide range of operators reflecting the instructions of the target assembly, and structured control-flow.

So as to enable the definition of a wide range of policies corresponding to various hardware and adversary capabilities, the definition of leakage is layered in two stages. First, its syntactic shape — formally defined on Figure 8 — is a tree whose structure reflects the program execution. The leakage ℓ_e of the evaluation of an expression is a tree whose leaves may be empty (•) or hold some value. This structure is fixed and corresponds to the "program

[3]For clarity of the exposition, only a fragment of the Jasmin language is presented. In particular, function calls and some kinds of loops are omitted. The actual formalization and proofs cover the full Jasmin language.

$\ell_e ::=$	$\bullet$	empty	$\ell ::=$	$\ell_e := \ell_e$	assignment
	$\mid v$	value		$\mid \mathsf{if}_b(\ell_e, \ell)$	conditional
	$\mid (\ell_e, \ldots, \ell_e)$	sub-leakage		$\mid \mathsf{while}_{\mathsf{t}}(\ell_e, \ell, \ell)$	iteration
				$\mid \mathsf{while}_{\mathsf{f}}(\ell_e)$	loop end
				$\mid \{\ell; \ldots; \ell\}$	sequence

Figure 8: Syntax of structured leakages

counter" leakage model. By leaking more or less precise values, the leakage model can be fine tuned: the second layer consists of two parameters: $\mathcal{A}^{\diamond}(v_1, \ldots, v_n)$ defines the leakage produced by the evaluation of the operator $\diamond$ applied to arguments $(v_1, \ldots, v_n)$; and $\mathcal{M}(p)$ defines the leakage produced by a memory access at address p.

Figure 9 presents the rules of the instrumented semantics of the Jasmin language. They follow a standard definition of a big-step semantics for a while language. There are three kinds of judgements: 1. $e \downarrow^{s}_{\ell_e} v$ corresponds to the evaluation of expression e in state s producing value v and leakage ℓ_e; 2. $d := v \downarrow^{s}_{\ell_e} s'$ corresponds to the assignment of value v into the left-value d in state s producing the updated state s' and leakage ℓ_e; 3. $i : s \Downarrow_{\ell} s'$ corresponds to the execution of instruction i starting in state s and ending in state s' while producing leakage ℓ. We sometimes use the notation $s \Downarrow^{\mathcal{A},\mathcal{M}}_{\ell} s'$ to make explicit the dependency of the semantic on the leakage model. For a given program i, an initial state s is said to be *safe* when there exists an execution from this state, ending in some final state s' and producing some leakage ℓ.

Note that constant and local variables do not leak; array accesses leak the value of the index; memory accesses leak according to $\mathcal{M}$; conditional expressions produce the leakage corresponding to the evaluation of the three sub-expressions (the guard as well as both branches) but do not leak the *value* of the guard; the leakage for arithmetic operators correspond to the leakage of the evaluation of their arguments followed by one of the computations of the operator (as defined by $\mathcal{A}$). The parameter defining the leakage of memory accesses is also used for defining the semantics of stores (not shown in the Figure).

3.2 Constant-Time Policies

Given this generic leakage model, the constant-time security property can be defined as usual. As the leakage model is parameterized, each instance of the parameters yields a particular policy.

Informally, a program is constant-time if two executions from two related input states yield equal leakage. The definition of leakage is parameterized by $\mathcal{A}$ and $\mathcal{M}$–and a relation φ on safe states — and gives rise to the notion of fine-grained constant-time.

Definition 3.1 (Fine-Grained Constant-Time). A program p is *constant-time* with respect to a relation φ on states and a leakage model $\mathcal{A}$ and $\mathcal{M}$, written $\mathsf{CT}^{\mathcal{A},\mathcal{M}}_{\varphi}(p)$, when

$$\forall s_1\ s_1'\ s_2\ s_2'\ \ell_1\ \ell_2,\ \ s_1\ \varphi\ s_2 \implies \begin{cases} p : s_1 \Downarrow_{\ell_1} s_1' \\ p : s_2 \Downarrow_{\ell_2} s_2' \end{cases} \implies \ell_1 = \ell_2.$$

Parameters:

$$\mathcal{M}(v), \mathcal{A}^{\mathrm{op}}(v_1, \ldots, v_n) \in \ell_e$$

Expression semantics:

$$\frac{}{c \downarrow^{s}_{\bullet} c} \qquad \frac{}{x \downarrow^{s}_{\bullet} s(x)}$$

$$\frac{e \downarrow^{s}_{\ell_e} z \quad s(a) = t}{a[e] \downarrow^{s}_{(\ell_e, z)} t[z]} \qquad \frac{e \downarrow^{s}_{\ell_e} p}{*e \downarrow^{s}_{(\ell_e, \mathcal{M}(p))} s[p]}$$

$$\frac{e \downarrow^{s}_{\ell_e} b \quad e_{tt} \downarrow^{s}_{\ell_{tt}} v_{tt} \quad e_{ff} \downarrow^{s}_{\ell_{ff}} v_{ff}}{\text{if } e \text{ then } e_{tt} \text{ else } e_{ff} \downarrow^{s}_{(\ell_e, \ell_{tt}, \ell_{ff})} v_b}$$

$$\frac{e_i \downarrow^{s}_{\ell_e^i} v_i \quad \mathrm{op}(v_1, \ldots, v_n) = v \quad \mathcal{A}^{\mathrm{op}}(v_1, \ldots, v_n) = \ell_e}{\mathrm{op}(e_1, \ldots, e_n) \downarrow^{s}_{((\ell_e^1, \ldots, \ell_e^n), \ell_e)} v}$$

Assignment semantics:

$$\frac{}{x := v \downarrow^{s}_{\bullet} s\{x \leftarrow v\}}$$

$$\frac{e \downarrow^{s}_{\ell_e} z \quad s(a) = t \quad t' = t\{z \leftarrow v\}}{a[e] := v \downarrow^{s}_{(\ell_e, z)} s\{a \leftarrow t'\}}$$

$$\frac{e \downarrow^{s}_{\ell_e} p}{*e := v \downarrow^{s}_{(\ell_e, \mathcal{M}(p))} s\{p \leftarrow v\}}$$

Instruction semantics:

$$\frac{}{\{\} : s \Downarrow_{\{\}} s}$$

$$\frac{i : s \Downarrow_{\ell_i} s_1 \quad \{c\} : s_1 \Downarrow_{\{\ell_c\}} s_2}{\{i; c\} : s \Downarrow_{\{\ell_i; \ell_c\}} s_2} \qquad \frac{e \downarrow^{s}_{\ell_e} v \quad d := v \downarrow^{s}_{\ell_d} s'}{d := e : s \Downarrow_{\ell_d := \ell_e} s'}$$

$$\frac{e \downarrow^{s}_{\ell_e} b \quad c_b : s \Downarrow_{\ell_c} s'}{\text{if } e \text{ then } c_{tt} \text{ else } c_{ff} : s \Downarrow_{\mathrm{if}_b(\ell_e, \ell_c)} s'}$$

$$\frac{e \downarrow^{s}_{\ell_e} ff}{\text{while } e \text{ do } c : s \Downarrow_{\mathrm{while_f}(\ell_e)} s}$$

$$\frac{e \downarrow^{s}_{\ell_e} tt \quad c, s \Downarrow_{\ell_c} s_1 \quad \text{while } e \text{ do } c : s_1 \Downarrow_{\ell_w} s_2}{\text{while } e \text{ do } c : s \Downarrow_{\mathrm{while_t}(\ell_e, \ell_c, \ell_w)} s_2}$$

Figure 9: Instrumented semantics

3.3 Instances

To illustrate the versatility of our definition, we present here example instantiations that correspond to the four leakage models of Section 2.4. In the BL model, arithmetic operations produce an empty leakage and memory accesses leak the pointer value. This can be expressed by the instances $\mathcal{A}_{\mathsf{BL}}$ and $\mathcal{M}_{\mathsf{BL}}$ which satisfy, for all operation $\diamond$, argument list $\mathbf{a}$, and pointer p:

$$\mathcal{A}^{\diamond}_{\mathsf{BL}}(\mathbf{a}) = \bullet$$
$$\mathcal{M}_{\mathsf{BL}}(p) = p.$$

3.3.1 Time-variable division. The TV leakage model captures that division instructions (signed and unsigned division and remainder computations) have a variable execution time. The arithmetic leakage function for this model is denoted by $\mathcal{A}_{\mathsf{TV}}$ and assumes that[4] $\mathcal{A}^{\div}_{\mathsf{TV}}(a, b) = (\log_2(a), \log_2(b))$. Similar equations hold for the other division-like operations. Other operations are modeled as constant-time and therefore produce an empty leakage, e.g., $\mathcal{A}^{\times}_{\mathsf{TV}}(a, b) = \bullet$.

This leakage model is a sound approximation that covers a wide range of actual architectures. For instance, this leakage model soundly approximates the behavior of the EPYC-7F52 processor, see Figure 2. For such a processor, leakage satisfies the following equation (and similar equations for other division-like operations): $\mathcal{A}^{\div}_{\mathsf{E}}(a, b) = \max(0, \log_2(\frac{a}{b}))$. This expresses that the execution time of the computation of a mod b is an affine function of the relative size of both arguments (when the dividend a is larger than the divisor b). Our model leaks more, and therefore absence of leakage with $\mathcal{A}^{\div}_{\mathsf{TV}}$ entails absence of leakage with $\mathcal{A}^{\div}_{\mathsf{E}}$.

3.3.2 Cache-line. The CL leakage model assumes that truncated addresses are leaked: $\mathcal{M}_{\mathsf{CL}}(p) = \lfloor \frac{p}{64} \rfloor$, where 64 is the (byte) granularity of cache lines. This model intuitively captures an adversary that can only witness cache-line conflicts. We use CL_{32} for a 32 bytes granularity.

3.3.3 Combining models. We can combine leakage functions. This leads to different models, including the BL leakage model, the TV leakage model, the CL leakage model (with 64 and 32 bytes), and the TV + CL leakage model.

4 COMPILER PRESERVATION OF FINE-GRAINED CONSTANT-TIME POLICIES

The Jasmin language gives the programmers precise control over low-level features: the compiler is predictable. Nonetheless, the compilation to assembly is a complex sequence of program transformations that rely on a handful of intermediate representations, as shown in Figure 10. In this section, we show that the Jasmin compiler preserves fine-grained constant-time policies. We first provide some relevant background.

4.1 Correctness

The Jasmin compiler preserves the behavior of programs.

THEOREM 4.1 (CORRECTNESS [2]). *For all p p',*

$$\mathsf{jasminc}(p) = \mathsf{OK}(p') \Rightarrow \forall s_{\mathsf{i}}\ s_{\mathsf{f}}, p : s_{\mathsf{i}} \Downarrow s_{\mathsf{f}} \Rightarrow p' : s_{\mathsf{i}} \Downarrow s_{\mathsf{f}}$$

where $\mathsf{OK}(p')$ *denotes that the compiler did not fail and returned the target program p'.*

Compiler correctness guarantees that trace properties carry from source to assembly programs. However, it does not provide any guarantee with respect to leakage.

[4]For readability, we simply note $\log_2(x)$ the integer part of the base-2 logarithm of $x + 1$; in the formal development we use the "complement" of this value that is computed by the LZCNT instruction.

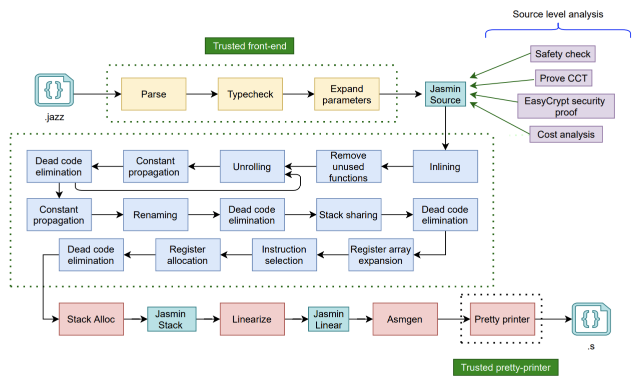

Figure 10: Compilation passes in the Jasmin compiler

4.2 Baseline instrumented correctness

Compilers typically do not preserve leakage. Indeed compilation reorders instructions, simplifies or even removes computations; it also introduces intermediate computations to implement high-level features that are not readily available in the target language. Nonetheless, previous work [15] has shown that leakage in the baseline model is usually transformed in a way that is statically known: leakage transformations follow in some way the program transformations performed by the compiler, and the correctness property can be strengthened to *instrumented correctness*. The compiler also produces a function F that describes how leakage is transformed for *all* executions.

THEOREM 4.2 (INSTRUMENTED CORRECTNESS [15]).

$$\forall p\ p'\ F,\ \ \mathsf{jasminc}(p) = \mathsf{OK}(p', F) \implies$$
$$\forall s_\mathrm{i}\ \ell\ s_\mathrm{f},\ \ p : s_\mathrm{i} \Downarrow_\ell s_\mathrm{f} \implies p' : s_\mathrm{i} \Downarrow_{F(\ell)} s_\mathrm{f}$$

where $\mathsf{OK}(p', F)$ *denotes that the compiler did not fail and returned the target program* p' *and the leakage transformer* F.

Preservation of baseline constant-time follows.

COROLLARY 4.3 (BASELINE CONSTANT-TIME PRESERVATION).

$$\forall p\ p'\ F,\ \ \mathsf{jasminc}(p) = \mathsf{OK}(p', F) \implies$$
$$\forall \varphi,\ \ \mathsf{CT}_\varphi^{\mathcal{A}_{\mathrm{BL}}, \mathcal{M}_{\mathrm{BL}}}(p) \implies \mathsf{CT}_\varphi^{\mathcal{A}_{\mathrm{BL}}, \mathcal{M}_{\mathrm{BL}}}(p').$$

4.3 Fine-Grained Instrumented Correctness

In this section, we extend instrumented correctness to fine-grained leakage models.

THEOREM 4.4 (FINE-GRAINED INSTRUMENTED CORRECTNESS).

$$\forall p\ p'\ F,\ \ \mathsf{jasminc}(p) = \mathsf{OK}(p', F) \implies$$
$$\forall s_\mathrm{i}\ \ell\ s_\mathrm{f}\ \mathcal{A}\ \mathcal{M},\ \ p : s_\mathrm{i} \Downarrow_\ell^{\mathcal{A},\mathcal{M}} s_\mathrm{f} \implies p' : s_\mathrm{i} \Downarrow_{F(\ell)}^{\mathcal{A},\mathcal{M}} s_\mathrm{f}.$$

The implementation of the compiler does not depend on the instance of the leakage model. Therefore, in the statement above, the function F that transforms the leakages only depends on the source and target programs. A single function accurately describes the transformation of leakages for all executions and in all leakage models.

This theorem entails that the Jasmin compiler preserves fine-grained constant-time security. Indeed, given two source executions with the same leakage ℓ, both corresponding executions at the target level have the same leakage $F(\ell)$.

COROLLARY 4.5 (FINE-GRAINED CONSTANT-TIME PRESERVATION).

$$\forall p\ p'\ F,\ \ \mathsf{jasminc}(p) = \mathsf{OK}(p', F) \implies$$
$$\forall \mathcal{A}\ \mathcal{M}\ \varphi,\ \ \mathsf{CT}_\varphi^{\mathcal{A},\mathcal{M}}(p) \implies \mathsf{CT}_\varphi^{\mathcal{A},\mathcal{M}}(p').$$

5 DEDUCTIVE ENFORCEMENT OF FINE-GRAINED CONSTANT-TIME POLICIES

(Fine-grained) constant-time policies are 2-safety properties and can be enforced using relational program logics, such as Relational Hoare Logic [16]. These logics manipulate judgements of the form:

$$c_1 \sim c_2\ :\ \varphi \Longrightarrow \psi$$

where c_1, c_2 are programs and φ is a relational pre-condition and ψ a relational post-condition. Both the pre-condition and the post-condition are interpreted as relations over the states of the two programs. Concretely, the interpretation of such a judgement is:

$$\forall s_1, s_1', s_2, s_2',\ \begin{cases} s_1\ \varphi\ s_2 \\ c_1 : s_1 \Downarrow s_1' \\ c_2 : s_2 \Downarrow s_2' \end{cases} \implies s_1'\ \psi\ s_2'$$

In other words, if we start the evaluation of c_1 and c_2 in two states that are in relation for the pre-condition ($s_1\ \varphi\ s_2$), the final states will be in relation for the post-condition ($s_1'\ \psi\ s_2'$). Note that the validity of a judgement is implicitly parametrized by an interpretation of operators. We write $\mathcal{A}, \mathcal{M} \models c_1 \sim c_2\ :\ \varphi \Longrightarrow \psi$ to reflect that a judgement is valid w.r.t. an interpretation of operators and memory leakage.

Relational Hoare Logic naturally captures information flow properties. For instance, consider a basic setting where every variable comes with a security label H or L. This induces an equivalence relation $=_{\{\mathrm{low}\}}$ on states. Then a program is non-interfering iff the following judgement is valid:

$$c \sim c\ :\ =_{\{\mathrm{low}\}} \Longrightarrow =_{\{\mathrm{low}\}}$$

A similar approach can be used to reason about side-channel leakage (see related work), at the cost of instrumenting programs to track their leakage. In the next paragraph, we sketch the instrumentation process and establish its correctness.

5.1 Instrumentation

The instrumentation of programs is generic and transforms every source program p into a new program $[p]$. p and $[p]$ have the same semantics except that $[p]$ accumulates the leakage generated by the evaluation of p in a special fresh program variable named leak.

The instrumentation relies on expressions extended with constructions $\mathsf{M}(e)$, $\mathsf{A}^\diamond(e, \ldots, e)$, $\emptyset$, and $e \uplus e$, as shown on top of Figure 11. The semantics of the $\mathsf{M}(\cdot)$ and $\mathsf{A}^\diamond(\cdot)$ constructions depends on the leakage model ($\mathcal{M}$ and $\mathcal{A}$). $\uplus$ operator is used for concatenation of leakages.

Instrumentation is defined first on expressions and then on instructions. The instrumentation $\{e\}$ of an expression e is shown in

Extended expressions:

$$\mathsf{Expr} ::= \ldots \mid \mathsf{M}(e) \mid \mathsf{A}^{\diamond}(e, \ldots, e) \mid \emptyset \mid e \uplus e$$

Translation of expressions:

$$\begin{aligned}
\{c\} = \{x\} &= \emptyset \\
\{*e\} &= \mathsf{M}(e) \uplus \{e\} \\
\{a[e]\} &= e \uplus \{e\} \\
\{\diamond(e_1, \ldots, e_n)\} &= \mathsf{A}^{\diamond}(e_1, \ldots, e_n) \uplus \{e_1\} \uplus \ldots \uplus \{e_n\} \\
\{\text{if } e \text{ then } e_{tt} \text{ else } e_{ff}\} &= \{e\} \uplus \{e_{tt}\} \uplus \{e_{ff}\}
\end{aligned}$$

Translation of instructions:

$$\begin{aligned}
[d := e] &= \mathsf{leak} := \{d\} \uplus \{e\} \uplus \mathsf{leak}; d := e \\
[\text{if } e \text{ then } c_{tt} \text{ else } c_{ff}] &= \mathsf{leak} := e \uplus \{e\} \uplus \mathsf{leak}; \\
&\quad\ \text{if } e \text{ then } [c_{tt}] \text{ else } [c_{ff}] \\
[\text{while } e \text{ do } c] &= \mathsf{leak} := e \uplus \{e\} \uplus \mathsf{leak}; \\
&\quad\ \text{while } e \text{ do } [c]; \mathsf{leak} := e \uplus \{e\} \uplus \mathsf{leak} \\
[i_1; \ldots; i_n] &= [i_1]; \ldots; [i_n]
\end{aligned}$$

Figure 11: Program instrumentation with explicit leakage

Figure 11. The instrumentation computes a new expression that will evaluate the leakage generated by e: if $e : s \downarrow_{\ell_e} v$ then $\{e\} : s \downarrow \ell_e$.

For array accesses $a[e]$ the leakage contains the index e and the leakage $\{e\}$ generated by its evaluation. The case of memory accesses is similar except that $\mathsf{M}(e)$ is leaked instead of e. In the case of operators, their leakages contain the leakage of their arguments and they also contain the leakage due to the operation. Remark that the leakage of an if expression[5] does not leak the value of the conditional expression e (only the leakage $\{e\}$ generated by e is leaked), this is possible because Jasmin compiler will compile this kind of expression using a conditional move instruction which is constant-time.

The bottom of Figure 11 provides the instrumentation for instructions $[c]$. The instrumentation of an assignment instruction $[d := e]$ is a sequence of two assignments, the first extends the variable leak with $\{d\}$ and $\{e\}$ and then do the assignment $d := e$. For conditional instructions, the leak variable is extended with the leakage $\{e\}$ generated by e but also with the value of the conditional itself (i.e., e). For the while loop, the instrumentation follows the same spirit; notice that the leak variable is updated once before the loop (to capture the leakage of conditional for the first iteration) and then at the end of each loop iteration.

5.2 Correctness of instrumentation

In this section, we provide the lemmas ensuring the correctness of instrumentation. The first lemma shows that the instrumented program correctly accumulates the leakage in the variable leak.

Lemma 1 (Correctness of the instrumentation)**.** For all program c, if its evaluation starting from a state s generates a leakage ℓ and a state s', then the evaluation of its instrumentation $[c]$ starting from the state s extended with ℓ_0 for the variable leak leads to the state s' extended with $\ell \uplus \ell_0$ for the variable leak:

$$\forall c\ s\ s'\ \ell\ \ell_0\ \mathcal{A}\ \mathcal{M}, c : s \Downarrow_{\ell}^{\mathcal{A},\mathcal{M}} s' \implies$$
$$[c] : s + \{\mathsf{leak} \leftarrow \ell_0\} \Downarrow^{\mathcal{A},\mathcal{M}} s' + \{\mathsf{leak} \leftarrow \ell \uplus \ell_0\}$$

where the notation $s + \{\mathsf{leak} \leftarrow \ell_0\}$ represents the state s extended with a fresh variable leak and its associated value ℓ_0.

The next lemma shows that fine-grained constant-time policies can be verified using relational Hoare logic.

Lemma 2 (Fine-grained constant-time, relationally)**.** If

$$\mathcal{A}, \mathcal{M} \models [c] \sim [c] \ :\ \varphi \wedge =_{\{\mathsf{leak}\}} \implies =_{\{\mathsf{leak}\}}$$

then $\mathsf{CT}_{\varphi}^{\mathcal{A},\mathcal{M}}(c)$.

The proof is a direct consequence of lemma 1 and of the interpretation of relational Hoare logic. It follows that any sound proof system for relational Hoare logic can be used for proving fine-grained constant-time.

6 IMPLEMENTATION

We have have implemented our approach in the Jasmin framework. Our implementation consists of:

- a Coq formalization of fine-grained leakage, and a formal proof that the Jasmin compiler preserves fine-grained constant-time policies;
- an OCaml implementation that extracts an EasyCrypt program from a Jasmin program;
- an OCaml implementation of an evaluator for testing constant-timeness.

The first two components of the implementation preserve the workflow for Jasmin programs:

- Jasmin programs are checked for safety and compiled;
- proofs of constant-timeness are carried on source Jasmin programs via an embedding into EasyCrypt[6]. Program instrumentation is performed during the embedding, in a way similar to what is presented in Section 5.

The last component adds an additional functionality (testing for constant-timeness), which is extremely helpful when dealing with fine-grained policies, which have considerably more complex proofs.

6.1 Coq formalization

We extended the work presented in [15] to reason about the fine-grained constant-time policies. The instrumented semantics are adapted to take into account the fact that each division-like operator $\diamond$ generates a leakage depending on its arguments $\mathcal{A}^{\diamond}$ (in [15] this leakage was assumed to be constant). As well, the leakage for memory access has been made generic to support weaker models (like CL). All the proofs of the compiler have been adapted and generalized over the leakage model. We have reused the leakage transformers of the prior work [15] with modifications to the ones that are used in the compiler passes like constant folding and instruction selection as they deal with propagation and lowering of operators. The correctness statements for each compiler pass and

[5]This is true for if expression not for if instruction.

[6]In principle it would be possible to verify constant-timeness within Coq using a formalization of relational Hoare logic, but this is orthogonal to the concerns of this paper.

their proofs are updated accordingly. The main theorems presented in 4.3 are stated once for the compiler as a whole, and their proofs are just corollaries of the correctness theorem.

Proof effort. The overall adaptation in the formalization made the Coq development grow from about 37×10^3 lines to 38×10^3 lines and required about four weeks of work.

6.2 Extraction to EasyCrypt & Jasmin evaluator

Jasmin provides different ways to extract programs to EasyCrypt. The first is used to prove functional correctness and cryptographic security of Jasmin programs; this is not affected by our work. The second is used to prove that Jasmin programs are constant-time. We have modified the latter to make it parametric in the leakage model. The extraction is done generically, independently of the model, then the users can specify their model in EasyCrypt. We provide the models presented in this paper, but the users are free to define their own models. This makes sense because the proof of preservation of constant-time is generic in the model.

Programming effort. The extraction to EasyCrypt is implemented in Ocaml, and the overall changes w.r.t. [15] amounts to around 50 lines of code. The evaluator is also implemented as an Ocaml wrapper; it uses automatically generated file (from the extraction of the Jasmin semantic defined in Coq), the wrapper represent represents around 200 lines of code.

6.3 Impact on compilation time

A natural question is whether the time for compilation and generation to EasyCrypt increases with the complexity of models. There is no noticeable overhead compared to [15], so we do not report any benchmark here.

7 EVALUATION

We evaluate the verification of fine-grained policies presented in this paper along different axes: its cost relative to the verification of the base-line policy; its effectiveness on actual examples extracted from existing code bases; its potential to support the design of secure and efficient implementations. More specifically, through several case studies, we study the following questions:

- what is the overhead of our generic approach for proving programs that are constant-time in the baseline model?
- what is the cost of proving programs constant-time with respect to other policies?
- how can formal tools supporting fine-grained policies contribute to the design of new implementations?

The first question is primarily a sanity check: indeed, one would like that our generic approach incurs minimal overhead in comparison to prior approaches. More specifically, we would like that programs that have a fully automated proof of BL constant-time in [15] also have an automated proof with our new pipeline, and programs that have an interactive proof of BL constant-time in [15] have a very similar interactive proof of constant-time in the new pipeline. We consider a set of examples from the Jasmin library, and validate this intuition.

The second question is more interesting because proofs in fine-grained models involve arithmetic and are thus more complex than proofs that focus on data-dependencies. We implement a set of examples that target fine-grained policies and prove or disprove that these examples satisfy their intended policies. As an indicative measure of the cost of verification, we report the number of lines in interactive proofs.

The last question illustrates how formal methods can be used in the early phases of development. To this end, we have developed and proved a generic version of the modulo which is secure in the TV model.

7.1 Impact on verifying the baseline policy

As a first case study, we consider Jasmin implementations that are secure in the BL model. Apart from the first implementations of the motivating example from Section 2.4 (ssl3_cbc_copy_mac_BL_BL and ssl3_cbc_copy_mac_TV_BL) that have been adapted for this work from OpenSSL, all implementations are taken from previous works [3, 4]. They consist of three versions of the ChaCha20 stream cipher [17] (a reference implementation and two optimized ones targeting specific vector instruction set extensions, namely AVX and AVX2); three versions of the Poly1305 authenticator; three versions of the Keccak1600 (SHA3 [18]) hashing algorithm (a reference implementation, an optimized one using AVX2 vector instructions, and an optimized one using scalar instructions only); two versions of MAC extraction based on the functions shown in Figure 4.

The results of this case study are reported in the first rows of Table 1. Here are a few observations. Eight out of the ten implementations can be proved secure in any of the considered policies in a proof script of seven lines only. The proof is a one-line call to a fully automatic tactic. The other lines are the statement of the theorem. The proof script for the vectorized version of Keccak1600 is much longer: this implementation uses in-memory tables, and its proof involves a lot of reasoning about pointers, similarly to what is done in the proof of functional correctness. This suggests possible improvements to the Jasmin programming language.

The last example also has a very short proof script (sixteen lines): since the precondition of the security statement involves the contents of the initial memory, the proof requires user interaction in a couple of places; it is nonetheless straightforward and mostly automatic.

In all cases, the proof can be carried once in the BL model, and the same script can be reused as-is in any weaker model (CL_{32} and CL_{64}). Also, for implementations that do not use time-variable instructions, proof extends without modification to the stronger TV and TV + CL models.

7.2 Verification effort of other policies

In this second case study, we consider implementations designed to be secure in non-baseline policies. They either target the TV model and ensure that time-variable operations only leak public information, or target one of the CL_{32} and CL_{64} models and make sure that only the least significant bits of pointers may be secret. We studied 10 cryptographic libraries; out of them, we found 8 libraries that contain such code, and we selected three examples that are the most representative or challenging. The corpus consists in four implementations of MAC extraction as explained in Section 2.4, one MAC verification, and the char2val routine used in base64 decoding.

Table 1: Compliance of examples with fine-grained policies. The table reports the size (lines of code) of each version of examples, and for each model whether it can be proved constant-time (number of lines of the proof) or if there is a counter-example (given in the supplementary material, marked with a ✘) witnessing a security violation.

Example	Implementation	Leakage model					
	size (loc)	CL_{32}	CL_{64}	BL	TV + CL_{32}	TV + CL_{64}	TV
ChaCha20 (ref)	396	7	7	7	7	7	7
ChaCha20 (AVX)	900	7	7	7	7	7	7
ChaCha20 (AVX2)	1006	7	7	7	7	7	7
Poly1305 (ref)	239	7	7	7	7	7	7
Poly1305 (AVX)	1065	7	7	7	7	7	7
Poly1305 (AVX2)	1037	7	7	7	7	7	7
keccak1600 (ref)	392	7	7	7	7	7	7
keccak1600 (AVX2)	446	361	361	361	361	361	361
keccak1600 (scalar)	469	7	7	7	7	7	7
ssl3_cbc_copy_mac_BL_BL	99	16	16	16	✘	✘	✘
ssl3_cbc_copy_mac_TV_BL	103	16	16	16	59	59	59
ssl3_cbc_copy_mac_BL_CL_{64}	82	✘	56	✘	✘	✘	✘
ssl3_cbc_copy_mac_TV_CL_{32}	89	153	156	✘	159	162	✘
ssl3_cbc_copy_mac_TV_CL_{64}	86	✘	59	✘	✘	90	✘
pmac_verify_hmac	78	118	118	✘	118	118	✘
coding_wolfSSL	34	✘	58	✘	✘	58	✘

All of them are new ports to Jasmin of existing code: the first five from OpenSSL and the last one from wolfSSL. Moreover, all of these examples are out of reach of the previous Jasmin pipeline.

The MAC verification (coined pmac_verify_hmac) accesses using secret dependent indices a 32-byte-aligned buffer of length at most twenty. Its security, therefore, relies on the assumption that the size of a cache line is at least 32 bytes. The base64 decoding excerpt (coding_wolfSSL) uses a table lookup at a secret dependent index: said table is 64-byte-aligned, size of 80 bytes and assumed to fit in two cache lines.

The sizes of the proofs corresponding to these examples are reported in the last six lines of Table 1. These proofs are generally longer than the ones for the baseline model: some arithmetic invariants about the arguments to time-variable operations and constraints on the base address and offsets for secret dependent memory accesses must be established.

As illustrated by the coding_wolfSSL case, which does not use time-variable operations, the proof script for the CL_{32} model can be reused without modification in the stronger TV + CL_{32} model.

Overall, the complexity of these proofs matches our expectations and is reasonable. For comparison, functional correctness proofs of ChaCha20 and SHA3 (to be found in earlier works) need hundreds and thousands lines respectively; the security proof of SHA3 needs tens of thousands of lines.

7.3 Security of MAC extraction

We now present in Section 7.3.1 some details of the proof of the ssl3_cbc_copy_mac implementations and in Section 7.3.2 that part of OpenSSL implementation of rotate_mac is insecure in the 32-byte cache line model and a verified patch.

7.3.1 Proving ssl3_cbc_copy_mac. We provide the EasyCrypt specification of some functions and insights to understand their proofs. The function is mainly a composition of the functions rotated_offset and rotate_mac since we have two implementations for each of them, this lead to four implementations.

The first function we consider is the rotate_mac_BL (top of Figure 5). This function is constant-time in the BL leakage model (and so in CL). This is captured by the following judgment:

$$\text{rotate\_mac} \sim \text{rotate\_mac} \;:\; =_{\{\text{leak, out, md\_size}\}} \Longrightarrow =_{\{\text{leak}\}}$$

This judgment states that if we start from two states in which the values of the variables leak, out, and md_size are equal, then the evaluations will end in two states where the value of the variable leak will be equal. In other words, if out and md_size are public, then the function is constant-time. out is a pointer that is public, which means its value (the address) will be equal on both sides, and it reveals nothing about the data stored at the address. The judgment requires no assumptions for the value of the variable rotate_offset; hence it can be secret dependent. The proof in EasyCrypt is fully automatic and uses an automated tactic based on dependency analysis. Similarly, we prove that the function rotate_offset_BL is secure in the BL model.

Next we prove that rotate_offset_TV is secure in the TV model. The pre-condition is:

$$\begin{aligned} &=_{\{\text{leak, md\_size, scan\_start}\}} \\ &\land (0 \le \text{mac\_start}\{1\} - \text{scan\_start}\{1\} < 256) \\ &\land (0 \le \text{mac\_start}\{2\} - \text{scan\_start}\{2\} < 256) \\ &\land 16 \le \text{md\_size}\{1\} \le 64 \end{aligned}$$

and the post-condition is simply $=_{\{\text{leak}\}}$. The notation x{1} refers to the value of the variable x in the left state while x{2} refers to its value in the right state. Remark that the variable mac_start is not

```
for (j = 0, i = 0; i < md_size; i++) {
    aux1 = rotated_mac[rotate_offset & ~32];
    aux2 = rotated_mac[rotate_offset | 32];
    mask = constant_time_eq_8(rotate_offset & ~32, rotate_offset);
    aux3 = constant_time_select_8(mask, aux1, aux2);
    out[j++] = aux3;
    rotate_offset++;
    rotate_offset &= constant_time_lt_s(rotate_offset, md_size);
}
```

Figure 12: Fixed C implementation of OpenSSL rotate_offset

required to be equal on both sides (i.e., the variable can be private), the precondition simply requires that its distance to scan_start is bounded (on both sides).

The proof of this statement requires about 25 lines of EasyCrypt code. It requires basic results on non-linear arithmetic. We use a shift by 23 to match the OpenSSL implementation, but a shift by 8 would suffice. One other interesting remark is that the original code for the shift was of the form (md_size >> 1) << 24. As md_size is even, it is equivalent to md_size << 23, but compilers can not infer it. The idea of writing it in this form by the OpenSSL developer was to prevent compiler from *optimizing* the code by removing introduced counter measure (i.e., replace ((md_size << 23) + rotate_offset) % md_size by (rotate_offset) % md_size, this replacement is functionally correct but does not preserve constant-time hyperproperty).

The function rotate_mac_CL is proved constant-time in the CL model, and it requires a stronger pre-conditions:

$$\begin{aligned}&=_{\{\text{leak, out, md\_size, rotated\_mac}\}}\\ &\wedge \text{rotated\_mac}\{1\}\%64 = 0\\ &\wedge 16 \le \text{md\_size}\{1\} \le 64\\ &\wedge 0 \le \text{rotate\_offset}\{1\} < \text{md\_size}\{1\}\\ &\wedge 0 \le \text{rotate\_offset}\{2\} < \text{md\_size}\{1\}\end{aligned}$$

In this implementation rotated_mac is a pointer to a buffer of length md_size, which should be public ($=_{\{\text{rotated\_mac}\}}$) and 64 byte aligned (it is the role of the caller of rotate_mac_CL to ensure this condition). The proof follows the intuition provided in Section 2.4. Since the specification also requires that rotate_offset{1}< md_size{1}, the specification of rotate_offset_TV needs to be extended to ensure that the result will satisfy this condition.

7.3.2 Verified countermeasure on rotating MAC with 32-byte cache line. OpenSSL implementation of rotate_mac_CL for 32-byte cache line model has a bug (see Figure 6 for original code). As the data will fit in two cache lines, there are two load operations within a loop. When trying to prove it against the CL_{32} model, we realized that this is incorrect and it is dependent on secret rotate_offset. We were able to easily create a counterexample using Jasmin evaluator. With a 64-byte aligned rotated_mac buffer, rotate_offset with value 31 touches the second cache line first and then the first. However, rotate_offset with value 63 touches the first cache line first and then the second.

Figure 12 shows the verified fix where we always access the first cache line and then the second cache line. Later, we select the correct value in constant-time. This incurs an overhead of 5.9% at ssl3_cbc_copy_mac function granularity. However, the overhead is negligeable in OpenSSL macro benchmarks.

7.4 Efficient Constant-Time Modulo

As a final example, we demonstrate how our approach can be used to guide the design of efficient constant-time code. The code for computing the modulus in constant-time manner (function rotated_offset_TV) works only because the value of the numerator (resp. the denominator) is small, less than 256 (resp. 64). In this section, we show that it is possible to implement a constant-time modulus without any requirement on the arguments except that the denominator needs to be public.

7.4.1 Jasmin implementation. Here is an implementation with the following assumptions: a is private, b is public, and we want to compute the (unsigned) remainder of the division of a by b. As the divisor is public, we know that the timing behavior of the hardware modulo instruction only depends on the size of the dividend. Therefore, before calling this instruction, we ensure that its first argument has a particular (public) size: its most significant bit must be set. This means that the modulo instruction is always called with its first argument in the range $[2^{63}; 2^{64} - 1]$.

To get a meaningful result, we compute an integer n such that $a' = b \cdot 2^n + a$ falls in the expected range. In this way, the computation of a' mod b will give the expected result.

The complete implementation in Jasmin is given in Figure 13. It relies on an auxiliary function lzcnt (lines 1–6) that counts the number of leading zeros of its argument and also returns a boolean flag telling if this number is null. We first compute (lines 11–17) lzb the number m of leading zeros of b and in the variable flag an integer whose value is one if a is already in the range and zero if a is not in the range but b is. A first attempt (lines 18–21) is to compute in variable dividend the value $b \cdot 2^{m-1} + a$. The use of the LEA instruction is an optimization that saves a copy. This value may be too small to fall in the target range; therefore a second attempt (lines 22–23) computes in variable temp2 the value $b \cdot 2^m + a$. This value is used as a dividend only if the addition did not overflow (line 24). In case b is in the range and a is not, then the result is a: in this case a dummy division of the maximal 64-bit unsigned integer is computed (lines 25–27) and the result accordingly corrected (line 30) using a conditional move.

7.4.2 Functional correctness. This implementation is correct (i.e., it always computes the modulo of its arguments). Correctness can be formally stated in EasyCrypt as follows:

$$\{\text{a} = a_0 \wedge \text{b} = b_0 \wedge \text{b} \neq 0\}\ \text{mod\_TV}\ \{\text{result} = a_0 \bmod b_0\}\,.$$

This is a Hoare triple with universally quantified logical variables a_0 and b_0 that allow referring to the initial values of the arguments. The proof boils down to showing that no overflow badly interferes with the computation and is about 60 lines long.

7.4.3 Constant-time security. This implementation is secure in the TV model, under the precondition that argument b is public and non-zero. Formally, the security is stated as follows, where the instrumentation of the mod_TV function is interpreted in the TV model:

$$\text{mod\_TV} \sim \text{mod\_TV}\ :\ =_{\{\text{leak,b}\}} \wedge b\{1\} \neq 0 \Longrightarrow =_{\{\text{leak}\}}$$

```
inline fn lzcnt(reg u64 x) ⟶ reg bool, reg u64 {
    reg u64 result;
    reg bool zf;
    _, _, _, _, zf, result = #LZCNT(x);
    return zf, result;
}
export fn mod_TV(reg u64 a, reg u64 b) ⟶ reg u64 {
    reg u64 flag, one, zero, dividend, modulo, result;
    reg u64 lzb, lzb_m1, b_lzb, b_lzb_m1, temp2;
    reg bool lzaz, lzbz, cf;
    flag = 0x1234;
    one = 1;
    zero = 0;
    lzbz, lzb = lzcnt(b);
    flag = zero if lzbz;
    lzaz, _ = lzcnt(a);
    flag = one if lzaz;
    lzb_m1 = #LEA(lzb - 1);
    b_lzb_m1 = b;
    b_lzb_m1 = b_lzb_m1 << lzb_m1;
    dividend = #LEA(b_lzb_m1 + a);
    b_lzb = b_lzb_m1 << 1;
    cf, temp2 = b_lzb + a;
    dividend = temp2 if ! cf;
    dividend = a if flag == 1;
    temp2 = 0xFFFFFFFFFFFFFFFF;
    dividend = temp2 if flag == 0;
    modulo = dividend % b;
    result = modulo;
    result = a if flag == 0;
    return result;
}
```

Figure 13: Generic constant-time modulus operation

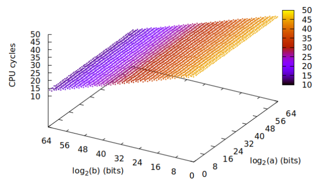

Figure 14: Timing behavior of the mod_TV function on one x86 microprocessor (same experimental setup as in Figure 2).

The proof methodology is similar to the other examples discussed earlier. The central argument is that the leakage produced by the execution of this function is a known function of b (hence independent of the value of the first argument a). Thanks to the simplicity of the control-flow structure of this program, the EasyCrypt machinery for computing the weakest preconditions can transform the program-verification task into a pure arithmetic formula. Discharging this proof is a bit tedious, as usual with machine arithmetic, due to the possible overflows. The proof script is about 130 lines long.

7.4.4 Experimental timing behavior. As illustrated in Figure 14, we experimentally observe that there is no longer any timing variation while changing the private input. The execution time of the whole function may still vary depending on the size of the public input. Notice on the right-hand-side plot that the execution times of the secure implementation range between 14 and 45 cycles, which is the same as the range of execution times of the lone hardware modulo instruction reported on the left of Figure 2 (for this particular microprocessor). Reproducing the same experiment on a different microprocessor (Intel Xeon E5-2687W) leads to the same conclusion: although the div instruction is time-variable, the execution time of the mod_TV function does not depend on the value of its first argument.

8 RELATED WORK

There is a large spectrum of tools for analyzing side-channel leakage [6, 25] of cryptographic implementations. Many tools, including ct-verif [11], flowtracker [28], virtualcert [8] and binsec/rel [23] explicitly target the baseline constant-time policy. These tools are supported by soundness claims. Only CacheD [32] considers a weaker leakage model where the cacheline is leaked. CacheD favors automation and precision over soundness and are therefore not supported by a soundness claim. As noted by Bernstein, no tool supports time-variable operations. There are many other tools, such as CacheAudit [24] or CacheFix [21], which use automated techniques to reason about cache behavior. However, these tools do not target a constant-time policy. In particular, they do not consider control-flow leakage and do not allow values to carry a security level. We refer to [6, 25] for a description of other tools. In addition, there are general-purpose tools that can also be applied to side-channel analysis. This is the case of Themis [22], which introduces QCHL, a quantitative variant of Cartesian Hoare Logic [29] and uses QCHL to reason about side-channels of Java bytecode. Another (earlier) instance is Blazer [5], which introduces a proof technique to reason about hypersafety and applies the proof technique to reason about side-channels of Java bytecode. Another line of work develops constraint-based methods for verifying relational properties of hardware, and applies these methods to reason about constant-time [30, 31]. There is also a large body of work that develops automated transformation methods for making programs constant-time, see e.g. [19, 20], or that develops frameworks that are secure-by-design: [34] features a language with mechanisms to control timing channels and a type system that quantitatively bound the information leakage of well-typed programs; [33] introduces a timing-channel aware ISA that serves as contract between software and hardware.

Constant-time analysis tools target source programs, intermediate languages, or low-level (assembly or binary) programs. Except in the latter case, compilers may turn non-constant-time programs that have been formally proved to be constant-time. This raises the question of proving that compilers preserve the constant-time property. Barthe, Grégoire and Laporte [14] develop a general approach based on CT-simulations for proving that a compiler preserves constant-time. CompCert-CT [9] is a formally verified (mild) variant of the CompCert compiler that preserves constant-time. The proof of preservation uses CT-simulations when needed but relies on the simpler property of leakage transformation when possible—for many compiler passes. The leakage transformer technique from [15] further refines this approach by introducing structured leakage and a syntax for leakage transformers. Our work leverages the benefits of leakage transformers to consider fine-grained information flow policies.

9 CONCLUSION

Proving that secure implementations are constant-time in fine-grained leakage models is difficult and requires complex reasoning about control flow and arithmetic invariants. We have extended the constant-time preservation proof of the Jasmin compiler to such fine-grained policies. Therefore, proofs of these policies can be done at the source level, using relational logic. We have demonstrated the flexibility of our approach on several challenging examples that were not proved before, and in the process fixed some leakage bug in OpenSSL.

An important direction for future work is to extend Jasmin with support for other CPU architectures, including ARM (Cortex), and other leakage models, including some newly proposed conceptual models (such as silent stores). We would then use this extension to prove preservation of constant-time and to formally verify constant-time of a broad corpus of cryptographic implementations.

REFERENCES

[1] Nadhem J Al Fardan and Kenneth G Paterson. 2013. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 526–540.

[2] 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, 1807–1823.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. 2020. The Last Mile: High-Assurance and High-Speed Cryptographic Implementations. In *2020 IEEE Symposium on Security and Privacy (S&P)*. 965–982. https://doi.org/10.1109/SP40000.2020.00028

[4] José Bacelar Almeida, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. 2019. Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. 1607–1622. https://doi.org/10.1145/3319535.3363211

[5] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. Decomposition instead of self-composition for proving the absence of timing channels. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 362–375. https://doi.org/10.1145/3062341.3062378

[6] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2021. SoK: Computer-aided cryptography. In *42nd IEEE Symposium on Security and Privacy*.

[7] Cécile Baritel-Ruet, François Dupressoir, Pierre-Alain Fouque, and Benjamin Gregoire. 2018. Formal Security Proof of CMAC and Its Variants. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. 91–104. https://doi.org/10.1109/CSF.2018.00014

[8] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel 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, Scottsdale, AZ, USA, November 3-7, 2014*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM, 1267–1279. https://doi.org/10.1145/2660267.2660283

[9] 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 (2020), 7:1–7:30. https://doi.org/10.1145/3371075

[10] Gilles Barthe, Francois Dupressoir, Benjamin Grégoire, Cesar Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. EasyCrypt: A Tutorial. In *FOSAD 2013 (Foundations of Security Analysis and Design {VII} - {FOSAD} 2012/2013 Tutorial Lectures)*. Bertinoro, Italy. https://doi.org/10.1007/978-3-319-10082-1_6

[11] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. 2016. Computer-Aided Verification for Mechanism Design. In *Web and Internet Economics - 12th International Conference, WINE 2016, Montreal, Canada, December 11-14, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10123)*, Yang Cai and Adrian Vetta (Eds.). Springer, 279–293. https://doi.org/10.1007/978-3-662-54110-4_20

[12] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. 2011. Computer-aided security proofs for the working cryptographer. In *Annual Cryptology Conference*. Springer, 71–90.

[13] Gilles Barthe, Benjamin Grégoire, Yassine Lakhnech, and Santiago Zanella Béguelin. 2011. Beyond provable security verifiable IND-CCA security of OAEP. In *Cryptographers' Track at the RSA Conference*. Springer, 180–196.

[14] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. IEEE Computer Society, 328–343. https://doi.org/10.1109/CSF.2018.00031

[15] Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2021. Structured Leakage and Applications to Cryptographic Constant-Time and Cost. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 462–476. https://doi.org/10.1145/3460120.3484761

[16] Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 14–25. https://doi.org/10.1145/964001.964003

[17] Daniel J Bernstein et al. 2008. ChaCha, a variant of Salsa20. In *Workshop record of SASC*, Vol. 8. 3–5.

[18] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2013. Keccak. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 313–314.

[19] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. 2021. Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 715–733. https://doi.org/10.1145/3460120.3484583

[20] 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

[21] Sudipta Chattopadhyay and Abhik Roychoudhury. 2018. Symbolic Verification of Cache Side-Channel Freedom. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 37, 11 (2018), 2812–2823. https://doi.org/10.1109/TCAD.2018.2858402

[22] Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic. 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, 875–890. https://doi.org/10.1145/3133956.3134058

[23] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2020. Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1021–1038. https://doi.org/10.1109/SP40000.2020.00074

[24] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, Samuel T. King (Ed.). USENIX Association, 431–446. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/doychev

[25] Jan Jancar. 2021. The state of tooling for verifying constant-timeness of cryptographic implementations. https://neuromancer.sk/article/26

[26] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. 2022. "They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks. In *IEEE SP*.

[27] Paul C Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*. Springer, 104–113.

[28] 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

[29] Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 57–69. https://doi.org/10.1145/2908080.2908092

[30] Klaus von Gleissenthall, Rami Gökhan Kici, Deian Stefan, and Ranjit Jhala. 2019. IODINE: Verifying Constant-Time Execution of Hardware. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1411–1428. https://www.usenix.org/conference/usenixsecurity19/presentation/von-gleissenthall

[31] Klaus von Gleissenthall, Rami Gökhan Kici, Deian Stefan, and Ranjit Jhala. 2021. Solver-Aided Constant-Time Hardware Verification. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 429–444. https://doi.org/10.1145/3460120.3484810

[32] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. 2017. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 235–252. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-shuai

[33] Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers. 2019. Using Information Flow to Design an ISA that Controls Timing Channels. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. 272–27215. https://doi.org/10.1109/CSF.2019.00026

[34] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. 2012. Language-Based Control and Mitigation of Timing Channels. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 99–110. https://doi.org/10.1145/2254064.2254078