Towards Efficient Verification of Constant-Time Cryptographic Implementations

LUWEI CAI, ShanghaiTech University, China
FU SONG*, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China, University of Chinese Academy of Sciences, China, and Nanjing Institute of Software Technology, China
TAOLUE CHEN, Birkbeck, University of London, UK

Timing side-channel attacks exploit secret-dependent execution time to fully or partially recover secrets of cryptographic implementations, posing a severe threat to software security. Constant-time programming discipline is an effective software-based countermeasure against timing side-channel attacks, but developing constant-time implementations turns out to be challenging and error-prone. Current verification approaches/tools suffer from scalability and precision issues when applied to production software in practice. In this paper, we put forward practical verification approaches based on a novel synergy of taint analysis and safety verification of self-composed programs. Specifically, we first use an IFDS-based lightweight taint analysis to prove that a large number of potential (timing) side-channel sources do not actually leak secrets. We then resort to a precise taint analysis and a safety verification approach to determine whether the remaining potential side-channel sources can actually leak secrets. These include novel constructions of taint-directed semi-cross-product of the original program and its Boolean abstraction, and a taint-directed self-composition of the program. Our approach is implemented as a cross-platform and fully automated tool CT-Prover. The experiments confirm its efficiency and effectiveness in verifying real-world benchmarks from modern cryptographic and SSL/TLS libraries. In particular, CT-Prover identify new, confirmed vulnerabilities of open-source SSL libraries (e.g., Mbed SSL, BearSSL) and significantly outperforms the state-of-the-art tools.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Theory of computation → Program analysis**; • **Security and privacy → Formal security models**; **Logic and verification**.

Additional Key Words and Phrases: Timing side-channel, constant-time cryptographic implementation, formal verification, taint analysis

ACM Reference Format:
Luwei Cai, Fu Song, and Taolue Chen. 2024. Towards Efficient Verification of Constant-Time Cryptographic Implementations. *Proc. ACM Softw. Eng.* 1, FSE, Article 46 (July 2024), 24 pages. https://doi.org/10.1145/3643772

1 INTRODUCTION

The security of contemporary software systems and communication heavily depends upon cryptographic implementations, which are the main focus of the current paper. Timing side-channel attacks [30, 54] can exploit secret-dependent execution time to fully or partially recover secrets

*Corresponding author

Authors' addresses: Luwei Cai, ShanghaiTech University, Shanghai, China, cailw@shanghaitech.edu.cn; Fu Song, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China and University of Chinese Academy of Sciences, Beijing, China and Nanjing Institute of Software Technology, Nanjing, China, songfu@ios.ac.cn; Taolue Chen, Birkbeck, University of London, London, UK, t.chen@bbk.ac.uk.

ACM 2994-970X/2024/7-ART46
https://doi.org/10.1145/3643772

even remotely, thus posing a severe threat to software security. Over the past few years, numerous timing side-channel vulnerabilities have been discovered, allowing adversaries to deduce secrets with very few trials. Notorious examples include the Lucky 13 attack that can remotely recover plaintext from the CBC-mode encryption in TLS and DTLS (due to an unbalanced branching statement [9]), and Brumley and Boneh's remote private key recovery attack against the sliding window exponentiation of the RSA decryption in OpenSSL [30, 31]. It is vital to implement effective countermeasures to protect cryptographic implementations.

There are different means to mitigate the risk of timing side-channels, for instance, via security-aware system (e.g., [56]) and architecture (e.g., [39]). A more effective approach is to come up with better software implementation to eliminate the root cause. Currently, there are two prevailing countermeasures, i.e., *constant-time* and *time-balancing* programming disciplines. The former requires that control flow and memory-access patterns of an implementation are independent of the secrets; the latter requires the execution time to be negligibly influenced by secrets which can be considered as a tradeoff between security and enforceability. Both countermeasures have been adopted in open-source cryptographic and SSL/TLS libraries such as NaCl [24], BearSSL [61], Mbed TLS [72], s2n-tls [16], and OpenSSL [4]. Writing constant-time or time-balancing implementations requires the use of low-level programming languages or compiler knowledge, and developers usually need to deviate from standard programming practices. Furthermore, their correctness depends on global properties across pairs of executions, hence is difficult to reason about. For instance, even though two protections against Lucky 13 were implemented in s2n-tls, the Lucky microseconds attack, a variant of Lucky 13, can remotely and completely recover plaintext from the CBC-mode cipher suites in s2n-tls, resulting in a complete recovery of HTTP session cookies and user credentials such as BasicAuth passwords [8]. Alas, timing side-channel attacks remain a live threat for cryptographic libraries after their discovery over 25 years ago [83], and it is essential to develop automated reasoning tools for formally verifying these countermeasures.

In this paper, we focus on the constant-time programming discipline as it represents a more fundamental solution and is more challenging to tackle. Numerous verification approaches have been proposed (cf. Section 6). A majority of them leverage (lightweight) static analysis, e.g., abstract interpretation, type system, taint analysis and (relational) symbolic execution which are sound (and usually efficient) but incomplete (i.e., false positives may occur or depth of explored paths is bound). Different from them, the self-composition [11] approach reduces constant-time verification to safety verification by building a program consisting of two copies of the given program and verifying whether the values of the low-security (public) variables in the two copies are identical provided that the public inputs are identical. Self-composition based approaches are sound and complete in theory, although in practice, the completeness may depend on verification tools to carry out challenging safety property checking on self-composed programs (cf. Section 5.1 for concrete examples).

Self-composition based approaches are normally considered as a heavyweight approach which is not scalable. The primary reason is that they need to deal with a self-composed program of quadratic sizes. As a result, many efforts have been made to ease safety verification of self-composed programs, including self-composition with lockstep execution of loops [68] for k-safety properties; self-composition with lockstep execution of both loops and branches (called cross-product) [11] for constant-time properties; type-directed self-composition [71] and lazy self-composition [81] for proving information flow properties. The commonality of these approaches is to simplify self-composed programs, reduce the number of safety checks, and/or keep variables from the two copies near each other.

Despite these efforts, a noticeable gap exists in verifying constant-time countermeasures in software engineering practice. To the best of our knowledge, ct-verif [11] is the only publicly

available self-composition based tool that is being actively deployed in the continuous integration of Amazon's s2n-tls library [67], but is significantly less efficient than lightweight static analysis approaches (e.g., taint analysis and type system), and often fails to prove constant-time implementations (cf. Section 5). In summary, the current status is that the user either chooses an incomplete approach but needs to tackle false positives, or chooses a (relatively) complete approach which is costly and may fail on a number of occasions.

The main purpose of the current paper is to make the self-composition based approaches scalable so they can be used to handle the verification of constant-time cryptographic implementations at the industry level. Our strategy towards both completeness and scalability is to take a stratified approach. Technically, starting from potential timing side-channel sources which can be identified straightforwardly (cf. Section 2.2), we devise two different taint analyses and a taint-directed cross-product, and gradually integrate them to resolve these potential sources, i.e., to determine whether they can actually cause information leakage.

The first taint analysis (cf. Section 4.1) leverages the inter-procedural, finite, distributive, subset framework (IFDS [64]), which is accelerated by propagating the data-flow facts sparsely [60]. This taint analysis is flow-, field- and context-sensitive, but path- and index-insensitive. It is often able to efficiently prove that a large number of potential (timing) side-channel sources do not actually leak secrets, leaving a relatively small number of them unresolved to the next step.

The second taint analysis (cf. Section 4.2) uses a novel *taint-directed semi-cross-product*, which reduces the flow-, context-, path-, field- and index-sensitive taint analysis problem to checking safety properties of the cross-product of the given program and its Boolean abstraction. The Boolean abstraction is used to track the required information flow from the secrets. This precise taint analysis would be able to further resolve many remaining potential side-channel sources. It is worth noting that our cross-product is taint-directed, namely, it is based on the taint information from the first taint analysis, which greatly reduces the number of safety checks and simplifies the product program, hence improving the verification efficiency.

Finally, to resolve the remaining (usually few) potential side-channel sources, we propose a *taint-directed cross-product* (cf. Section 4.3), which reduces the constant-time security problem to the safety problem of the cross-product of the program where taint information is also used to reduce the number of safety checks and simplify the cross-product program.

We implement our approaches as a fully automated, cross-platform tool CT-Prover for verifying (optimized) LLVM IR implementations. To evaluate CT-Prover, we collect 87 real-world implementations from modern cryptographic and SSL/TLS libraries, as well as fixed-point arithmetic libraries. These benchmarks include cryptographic utilities, arithmetic operations, public and private key cryptography, and algorithms for encryption, decryption, message authentication code, and digital signature. The experiment results confirm the effectiveness and efficiency of our approach. In particular, CT-Prover (dis)proves all the (non-)constant-time implementations and find new vulnerabilities in open-source libraries Mbed SSL and BearSSL, and is typically significantly faster than state-of-the-art tools.

In summary, we make the following main contributions:

- We provide a novel synergy of taint analysis and self-composition, improving the efficiency and scalability of verification of constant-time cryptographic implementations;
- We develop a fully automated tool to support the verification for production software at the industrial level;
- We conduct an extensive evaluation on a large set of real-world programs, and identify new, confirmed timing side-channel vulnerabilities of open-source SSL libraries.

Expressions:	e	::=	$n \mid x \mid e_1 \odot e_2 \mid x[y]$
Statements:	p	::=	$\texttt{skip} \mid x := e \mid x[y] := z \mid \texttt{assert}\ e \mid \texttt{assume}\ e \mid p;\ p \mid \texttt{while}\ x\ \texttt{do}\ p\ \texttt{od}$
		|	$\texttt{if}\ x\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{fi} \mid x_1, \cdots, x_m := f(y_1, \cdots, y_n)$
Procedures:	fn	::=	$\texttt{def}\ f(x_1, \cdots, x_n)\{p;\ \texttt{return}\ y_1, \cdots, y_m;\}$
Programs:	P	::=	fn^+

Fig. 1. The syntax of WHILE.

Outline. Section 2 presents the background of the work, including the WHILE language and basics of constant-time security. Section 3 gives motivating examples and an overview of our approach. Section 4 presents the details of our approaches. Section 5 reports the experiment results. Section 6 discusses the related work. The paper is concluded in Section 7.

2 PRELIMINARIES

Our constant-time verification tool works over programs in LLVM intermediate representation (IR). There are three main reasons: (1) LLVM IR is a low-level architecture-independent language so verification can be performed on optimized LLVM IR programs, (2) it is more convenient to verify and debug LLVM IR programs than binary executables, and (3) the verified compiler CompCert offers constant-time preserving compilation [22] and Binsec/Rel [36, 38] offers bug-finding and bounded verification for binary executables. However, for clarity, we use the WHILE language enriched with arrays, assert/assume statements and procedures, to define the notion of constant-time security and formalize our verification approach.

2.1 The WHILE Language

Syntax. The syntax of the WHILE language is given in Figure 1. A WHILE program consists of a sequence of procedures, one of which is the `main` procedure as the entry of the program. A procedure contains a sequence of formal arguments and a sequence of statements followed by a `return` statement. Note that in our WHILE language, the `return` statement can return a tuple of values which is required for constructing cross-products. For each procedure f, we denote by $\mathbb{X}_f$ the set of variables used by f, including its formal arguments.

Statements include `skip` statements, `assert` and `assume` statements, sequential statements, `if-then-else` and `while-do` statements, assignments, and procedure calls. As in ct-verif [11], we include the `assert` and `assume` statements to simplify the reduction to safety checking. We assume that each statement is annotated by a distinct label ℓ. Expressions include constants, variables, arithmetic operations and array accesses. We use $\odot$ to range over binary operators which are deterministic and side-effect free. (Unary operators can be defined similarly and are not presented here.) W.l.o.g., We assume that WHILE programs are given in the single-static assignment (SSA) form, array-read $x[y]$ cannot be used as sub-expressions and all identifies in a program are distinct.

Semantics. Fix a WHILE program P. Let $\mathbb{X}$ be the set of variables of P, $\mathbb{L} \subseteq \mathbb{X} \cup (\mathbb{X} \times \mathbb{N})$ be the set of locations comprising scalar variables and pairs (x, i) of array variables x and indices i, and $\mathbb{V}$ be the set of possible values of variables. A *state* $s : \mathbb{L} \to \mathbb{V}$ is a mapping from locations l to values $s(l)$, namely, it maps variables x (resp. array elements $x[i]$) to values $s(x)$ (resp. $s(x, i)$). An *initial state* s_0 is a mapping that only gives the values of the input variables of P, i.e., the parameters of the `main` procedure. The update of a state s is written as $s[l \mapsto n]$. We define $\perp$ as a distinguished *error state* at which the execution of the program is disabled. We denote by $s(e)$ the value of the expression e in the state s, i.e., $s(n) = n$, $s(e_1 \odot e_2) = s(e_1) \odot s(e_2)$, and $s(x[y]) = s(x, s(y))$.

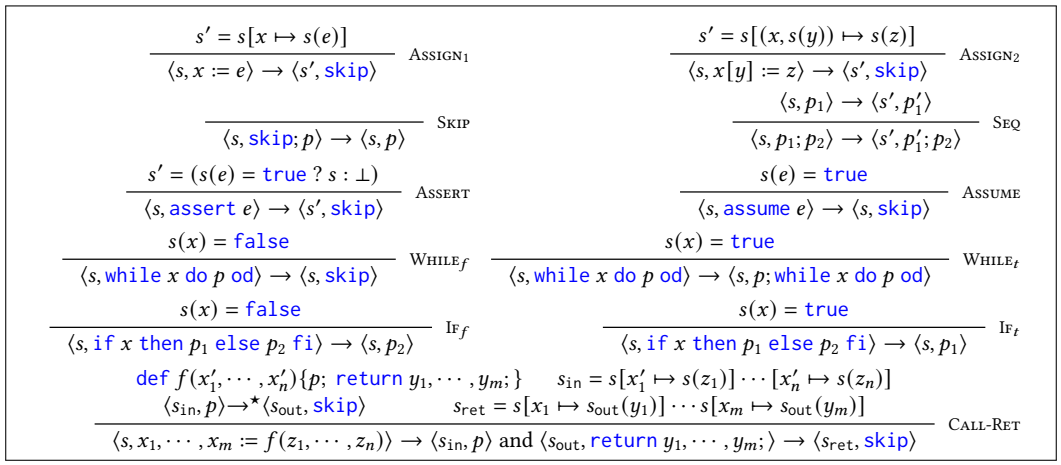

Fig. 2. The operational semantics of the While program.

A *configuration* c is a pair $\langle s, p\rangle$ consisting of a state s and a statement p to be executed. An *initial configuration* c_0 is a pair $\langle s_0, p\rangle$ such that s_0 is an initial state and p is the statement of the `main` procedure excluding the ending `return` statement. The semantics of the program P is defined as a transition relation $c \to c'$ between two configurations. We denote by $\to^\star$ the reflexive and transitive closure of the relation $\to$. The transition relation $c \to c'$ is given in Figure 2. For the sake of simplicity, array bounds are not checked in our semantics, and we assume that they are checked by using `assert` statements, thus executions are stuck on the error state when indices are out of the range of the array.

An *execution* ρ of the program P is a sequence of configurations $c_0 c_1 \cdots c_n$ such that c_0 is an initial configuration and $c_i \to c_{i+1}$ for every $0 \le i < n$. An execution ρ is *safe* if it does not stick on a configuration $\langle \bot, \cdot\rangle$ with the error state $\bot$, and is *complete* if it ends with a configuration $\langle \cdot, \texttt{skip}\rangle$. The program P is *safe* if all the executions are safe. We remark that in this work, we assume that programs always terminate (i.e., either complete or stick on), because we focus on cryptographic implementations. Termination can be checked by tools, e.g., CPAchecker [25] and T2 [29].

2.2 Constant-Time Security

In practice, execution time variations can create timing side-channel in various forms: (1) unbalanced branching statements may expose the information of the branching condition, (2) non-constant loops may expose the information of the loop condition, (3) memory access patterns (cache hits and misses) may expose the information of the memory address (i.e., indices of arrays in the While programs) accessed in load and store instructions, (4) time-variant instructions (e.g., integer divisions) in some architectures may expose the information of operands, and (5) micro-architectural features (e.g., Spectre [53] and Meltdown [57]) may break conventional constant-time guarantees. In this work, we consider the first three timing side-channels and use the common leakage model [11, 26, 36, 83] to characterize constant-time security. (The last two timing side-channels are not considered because they are architecture-dependent while we consider LLVM IR which is architecture-independent.) Note that our methodology is generic and could be adapted to handle the fourth timing side-channel by listing all the timing-sensitive operations [11]. Detection, verification and mitigation techniques of the fifth type of timing side-channels have been studied (cf. [33] for a survey), most of which assume that programs are constant-time without micro-architectural features, and thus are orthogonal to our work.

Definition 2.1 (Constant-time Leakage Model). Given a configuration $c = \langle s, p\rangle$ such that $s \neq \bot$, the observation $O(c)$ is defined as follows.

(1) if p is a branching statement `if` x `then` p_1 `else` p_2 `fi`, then $O(c) = s(x)$, namely, the value of the branching condition x is observable to the adversary;
(2) if p is a loop statement `while` x `do` p' `od`, then $O(c) = s(x)$, namely, the value of the loop condition x is observable to the adversary;
(3) if p is an assignment $z := y[x]$, then $O(c) = s(x)$, namely, the value of the index x in the load instruction is observable to the adversary;
(4) if p is an assignment $y[x] := z$, then $O(c) = s(x)$, namely, the value of the index x in the store instruction is observable to the adversary,
(5) if p is an sequential statement $p_1; p_2$, then $O(c) = O(\langle s, p_1\rangle)$, namely, only the executing instruction p_1 is considered (Note that p_2 will be considered in a subsequent configuration);
(6) otherwise $O(c) = \epsilon$, where ϵ denotes an empty observation.

For each statement p with label ℓ (denoted by $\ell : p$) as per Definition 2.1(1)-(4) where x is the operand or condition, (ℓ, x) is called a *potential (timing) side-channel source*. Intuitively, the value of x at label ℓ is observable to the adversary.

An execution $\rho = c_0 \cdots c_n$ yields the observation $O(\rho) = O(c_0) \cdots O(c_n)$. Two executions ρ_1 and ρ_2 are *indistinguishable* (to the adversary with respect to the leakage model O) if $O(\rho_1) = O(\rho_2)$. It is easy to see that two indistinguishable executions ρ_1 and ρ_2 must have the same control flow, i.e., they execute the same conditional branches and iterations of loops, thus the sequences of executed statements are the same. This observation is utilized to define cross-product [11], i.e., self-composition with lockstep execution of both loops and branches, namely, the copies share the same control flow.

Given a program P, we assume that the input variables $\mathbb{X}^{in} \subseteq \mathbb{X}_{\mathtt{main}}$ are partitioned into *public* input variables $\mathbb{X}^{in}_l$ and *secret* input variables $\mathbb{X}^{in}_h$. These sets are to be annotated by users. (For the sake of presentation, an input array variable should be annotated by either public or secret, meaning that all the elements of the array are public or secret.) In our implementation, we provide API wrappers to precisely annotate elements of input arrays and fields of input structures. The adversary knows the implementation details of the program and has access to the values of public input variables at runtime, but does not have any direct access to the values of other variables. The goal of the adversary is to infer the information of secret input variables by analyzing observations from executions.

Given a set of variables $X \subseteq \mathbb{X}$, two states s_1 and s_2 are *X-equivalent*, written as $s_1 \simeq_X s_2$, if for every scale variable $x \in X$, $s_1(x) = s_2(x)$ and for every array variable $x \in X$ and possible index $i \in \mathbb{N}$, $s_1(x, i) = s_2(x, i)$. Two configurations c_1 and c_2 are *X-equivalent*, written as $c_1 \simeq_X c_2$, if their states are *X-equivalent*. For a pair of executions $\rho = c_0 c_1 \cdots c_n$ and $\rho' = c'_0 c'_1 \cdots c'_{n'}$, $\rho \simeq_X \rho'$ denotes that for every $0 \leq i \leq \min(n, n')$, $c_i \simeq_X c'_i$.

Definition 2.2 (Constant-time Security [11]). A safe program P is *(constant-time) secure* if for any pair of complete executions $\rho = c_0 c_1 \cdots c_n$ and $\rho' = c'_0 c'_1 \cdots c'_{n'}$,

$$(c_0 \simeq_{\mathbb{X}^{in}_l} c'_0) \Rightarrow O(\rho) = O(\rho').$$

Intuitively, the safe program P is secure if, for any pair of complete executions that have the same public input values (i.e., the values of public input variables), their observations are the same, meaning that secret inputs are not distinguishable from the observations. Otherwise, there must exist a side-channel source (ℓ, x), namely, the values of the variable x at label ℓ differ between two configurations c_i and c'_i for some i. Thus, a potential timing side-channel source (ℓ, x) does not necessarily leak the secrets (i.e., x is secret-independent), but leaks the secrets when x is

```
uint64_t fixfrac(char* frac) {
    uint64_t pow10_LUT[20] = {0x1999999999999999, ...,0x0000000000000000};
    uint64_t pow10_LUT_extra[20] = {0x99, ..., 0x2f};
    uint64_t result = 0;          uint64_t extra = 0;
    for(int i = 0; i < 20; i++) {
        if(frac[i] == '\0') {  break;  }
        uint8_t digit = (frac[i] - (uint8_t) '0');
        result += ((uint64_t) digit) * pow10_LUT[i];
        extra  += ((uint64_t) digit) * pow10_LUT_extra[i];
    }
    ... }
```

Fig. 3. Fragment of the function `fixfrac` taken from the libfixedtimefixedpoint library.

secret-dependent. The security of unsafe programs is undefined, because they are stuck in the error state. (Note that the safety of a program can be verified using standard verification techniques and tools, e.g., SMACK [63].)

Standard library functions `malloc`, `free`, `memcpy` and `memset` may be used in cryptographic implementations. To handle them, following [11], we assume that the address and the length used in those functions are observable to the adversary, and the return of `free` is secret-independent.

3 MOTIVATION AND OVERVIEW

In this section, we present two motivating examples and an overview of our approach.

3.1 Motivating Examples

Example 1. Figure 3 shows a fragment of the function `fixfrac`, a fixed-point numeric operation provided by the library libfixedtimefixedpoint [13]. Given a digit string `frac` whose length is no more than 20, it computes a 64-bit number which corresponds to

$$\texttt{atoi}(\texttt{frac} + \texttt{padding})/10^{20}) * 2^{64}$$

where `frac+padding` is a digit string obtained from `frac` by padding some 0's such that the length is 20, and `atoi` coverts a digit string into the corresponding integer. The function `fixfrac` is invoked by the function `fix_pow`(x, y) which computes x^y over the fixed-point numbers x and y.

There are five potential side-channel sources in this code snippet, i.e., $(6, \texttt{frac[i]} == \text{`\textbackslash 0'})$, $(6, \texttt{i})$, $(7, \texttt{i})$, $(8, \texttt{i})$ and $(9, \texttt{i})$. We can observe that `i` is secret-independent, thus the last four pairs are not side-channel sources. However, it is non-trivial to determine the first potential side-channel source $(6, \texttt{frac[i]} == \text{`\textbackslash 0'})$. If it is, the information of the secret inputs x and y can be inferred by the adversary via timing side-channels.

This example cannot be proved by the self-composition based approach ct-verif [11] unless the loop is unrolled or the following loop invariant is added:

$$\exists i_{\max} \cdot 0 \leq i < i_{\max} \leq 20 \wedge \text{frac}\,[i_{\max}] == 0 \wedge \forall j . 0 \leq j \leq i_{\max} \Rightarrow \text{public}(\text{frac}[j]).$$

However, loops may be not statically bounded and providing such loop invariants manually is non-trivial, which means that in practice, the self-composition based approach is not fully automatic. In contrast, taint analysis is useful here. Indeed, `frac` is secret-independent.

Example 2. Figure 4 shows a simplified fragment of the function `crypto_stream_chacha20_ref` taken from the libsodium library, a portable, cross-compilable and installable fork of NaCl with an extended API to improve usability. This function implements the ChaCha20 stream cipher [23].

In the fragment of `crypto_stream_chacha20_ref`, variable `c` points to a plaintext to be encrypted, `clen` is the length of the plaintext, `n` points to an initialization vector, and `k` points to a

```
int crypto_stream_chacha20_ref(unsigned char *c, unsigned long long clen,
                               const unsigned char *n, const unsigned char *k){
    uint32_t ctx[16];
    chacha_keysetup(ctx, k);      // store k from ctx[0] to ctx[11];
    chacha_ivsetup(ctx, n, NULL); // store IV and counter into the rest;
    chacha_encrypt_bytes(ctx, c, c, clen);
}
static void chacha_encrypt_bytes(uint32_t *x, const u8 *m, u8 *c,
                                 unsigned long long bytes){
    j12 = x[12];
    if (!j12) {...}
}
```

Fig. 4. Simplified fragment of the function `crypto_stream_chacha20_ref` taken from the libsodium library.

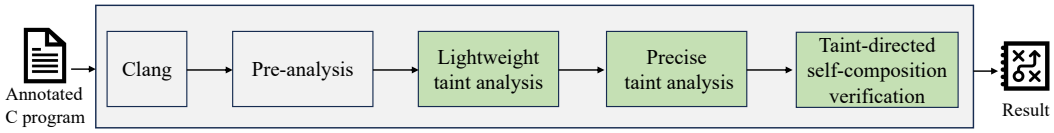

Fig. 5. Overview of our approach

private key. The key `k` is stored at the first 12 positions of the buffer `ctx`, the initialization vector and a counter (initialized as NULL) are stored at the rest 4 positions of the buffer `ctx`.

There is one potential side-channel sources in this simplified fragment, i.e., (11, `!j12`). It is non-trivial to determine this potential side-channel source, as the value of `j12` is `x[12]` while the buffer `x` contains the secret key `k`. If it is, the information of the secret key `k` can be inferred by the adversary via timing side-channels.

This example cannot be proved by an index-insensitive taint analysis, because `ctx` contains both secret-dependent and secret-independent contents, namely, `ctx[0–11]` and `ctx[12–15]`. Any index-insensitive taint analysis will conservatively taint the whole buffer. In contrast, an index-sensitive taint analysis (e.g., our precise taint analysis) would work in this example.

These examples reveal that static analysis (e.g., taint analysis) and self-composition based approaches may have complementary strengths even without efficiency considerations. Our method precisely takes advantage of their respective strength for which we provide an overview below.

3.2 Approach Overview

An overview of our approach is shown in Figure 5. In general, for a given annotated C program, the tool either outputs *proved*, suggesting that the program is secure, or outputs (potential) side-channel sources and corresponding execution traces for vulnerability localization and repair.

Our approach works as follows. First, the input program is translated into LLVM intermediate representation (IR) with annotations using Clang, the front-end of LLVM. Second, we compute the call graph, interprocedural control-flow graph, points-to information and definition-use chains required by the subsequent steps via a pre-analysis. Third, a lightweight taint analysis (cf. Section 4.1) is performed by leveraging the inter-procedural, finite, distributive, subset (IFDS) framework [64]. Often it is able to determine a large number of potential side-channel sources, leaving few potential side-channel sources unresolved. Finally, we resort to a precise taint analysis (cf. Section 4.2) and a taint-directed self-composition verification (cf. Section 4.3) to resolve the left-over potential side-channel sources. The precise taint analysis reduces the flow-, context-, path-, field- and index-sensitive taint analysis problem to checking safety properties of a cross-product of the given program and its Boolean abstraction which tracks the required information flow from the secrets.

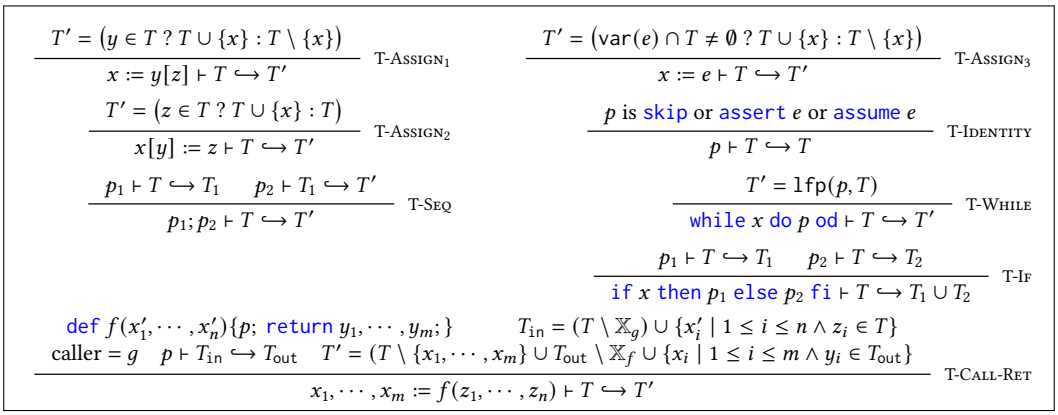

Fig. 6. Taint inference rules for the While language.

The taint-directed self-composition consists of two copies of the original program which is a new variant of self-composition. Remarkably, the taint information is utilized in both cross-product constructions to simplify the resulting program and reduce the cost of safety checks. By combining lightweight taint analysis and heavyweight safety verification, the overall approach brings the best of three worlds: efficiency, soundness and theoretical completeness.

Consider the motivating examples. The lightweight taint analysis is able to prove that all the five potential side-channel sources in Example 1 actually do not leak secrets, so the next two steps are not needed. In Example 2, the lightweight taint analysis fails to determine the potential side-channel source $(11, \texttt{!j12})$, thus the subsequent analyses have to check $(11, \texttt{!j12})$, which can be resolved by the precise taint analysis. (The final step is thus not needed.) We remark that the precise taint analysis may fail to prove some constant-time implementations meaning that it is sound but incomplete. For instance, when the secret k is involved in the computation of a potential timing side-channel source x (e.g., $x = k \oplus p \oplus k$ where $\oplus$ is Exclusive-OR), and the value of x is independent upon the secret k, the precise taint analysis will raise a false positive.

4 METHODOLOGY

In this section, we present the details of the three key components, i.e., lightweight taint analysis, precise taint analysis and taint-directed self-composition.

4.1 Lightweight Taint Analysis

In this subsection, we present a lightweight taint analysis which is designed to be flow-, field- and context-sensitive, but path-and index-insensitive, for a balance of efficiency and precision. Often it is able to prove that a large number of potential side-channel sources do not leak secrets, leaving few potential side-channel sources unresolved.

Taint source. Fix a safe program P. The taint source is the set $\mathbb{X}_h^{in}$ of its secret input variables, each of which is a taint fact. We remark that although our implementation supports the element-wise annotation of input array variables, this taint analysis is index-insensitive. Thus, if any element of an input array variable is annotated by secret, the array variable is regarded as secret, i.e., all the elements of the array are tainted.

Taint inference rule. The taint inference rule is given by the transfer function of the form

$$p \vdash T \hookrightarrow T',$$

where p is a statement, T and T' are sets of taint facts. The transfer function $p \vdash T \hookrightarrow T'$ means that the execution of the statement p with the set of taint facts T results in the set of taint facts T'.

The taint inference rules of the WHILE language are given in Figure 6, where $\texttt{var}(e)$ denotes the set of variables involved in the expression e, and $\texttt{lfp}(p, T)$ is recursively defined as follows:

$$\texttt{lfp}(p, T) = \begin{cases} T, & \text{if } p \vdash T \hookrightarrow T; \\ T \cup \texttt{lfp}(p, T'), & \text{otherwise, where } p \vdash T \hookrightarrow T'. \end{cases}$$

These rules are standard, which, intuitively, propagate taints from the right-hand side variables to the left-hand side variable. For example, rule [T-ASSIGN$_1$] expresses that if the array y is tainted, then the loaded array element is tainted. Rule [T-ASSIGN$_2$] expresses that if a tainted value is stored in an array, then the array is tainted. (Recall that our taint analysis is index-insensitive.) Rule [T-ASSIGN$_3$] expresses that if any involved variable of an expression e is tainted, then the result of the expression e is also tainted. Note that if the left-hand side is a scalar variable (i.e., rules [T-ASSIGN$_1$] and [T-ASSIGN$_3$]), we perform a strong update. Rule [T-WHILE] computes the least fixed point by applying the operator $\texttt{lfp}$ which always terminates, because the sequence of sets of taint facts during $\texttt{lfp}(p, T)$ is ascending w.r.t. the order $\subseteq$. Rule [T-CALL-RET] is much involved. The set $T_{\texttt{in}}$ of input taint facts at the call-site is obtained by passing actual arguments of the caller g to the formal parameters of the callee f after filtering out the taint facts of the local variables $\mathbb{X}_g$ of the caller g. The body p of the callee is analyzed using the set $T_{\texttt{in}}$ of input taint facts, leading to the set $T_{\texttt{out}}$ of output taint facts. The set $T_{\texttt{out}}$ of output taint facts is merged with the set T of taint facts at return-site after filtering out the taint facts of the local variables $\mathbb{X}_f$ of the callee f and the actual return variables are updated accordingly.

IFDS-based taint analysis. We leverage the inter-procedural, finite, distributive, subset (IFDS) framework to implement the lightweight taint analysis. The time complexity and space complexity of the vanilla IFDS algorithm are $\mathbf{O}(|E| \cdot |D|^3)$ and $\mathbf{O}(|E| \cdot |D|)$, respectively, where $|E|$ is the number of edges in the interprocedural control-flow graph and the size $|D|$ of the domain is the number of all possible taint facts, i.e., $|\mathbb{X}|$. The time complexity will increase sharply with $|\mathbb{X}|$ in practice [55] which is significant for some cryptographic implementations due to the following reasons.

On the one hand, typically, the input of a cryptographic algorithm consists of a key and plaintext; the key is secret and thus tainted, and the key and plaintext are tightly coupled in the computation. For instance, the AES algorithm has multiple modes, with the smallest key size being 128 bits (an array with 16 elements) and the smallest encryption process being 10 rounds each of which has four transformations. On the other hand, the SSA form introduces a number of temporary variables. Consequently, a large number of taint facts are propagated during the taint analysis. To mitigate this issue, inspired by the sparse data-flow analysis [60], we improve the classic IFDS framework by directly propagating taint facts of scalar variables via data flow instead of control flow. More specifically, if a scalar variable x is tainted, this taint fact is directly propagated to the statements where x is used, using the def-use chains. It avoids the propagation of taint facts of scalar variables for many statements, hence improving efficiency in practice.

Hereafter, for every label ℓ of a statement p, we denote by T_ℓ the set of taint facts at label ℓ (i.e., before the execution of the statement p), obtained by applying the IFDS-based taint analysis. Since the IFDS-based taint analysis is sound, it is straightforward to have that

LEMMA 4.1. *For any potential side-channel source (ℓ, x) and pair (c_0, c_0') of initial configurations such that $(c_0 \simeq_{\mathbb{X}_I^{in}} c_0')$, if x is not tainted at the label ℓ, i.e., $x \notin T_\ell$, then the values of x are the same at the label ℓ in any pair of complete executions $\rho = c_0 c_1 \cdots c_n$ and $\rho' = c_0' c_1' \cdots c_{n'}'$.*

Proof sketch. For a pair of complete executions $\rho = c_0 c_1 \cdots c_n$ and $\rho' = c'_0 c'_1 \cdots c'_{n'}$ with $c_0 \simeq_{\mathbb{X}_l^{in}} c'_0$, suppose that the values of x are different at the label ℓ. It suffices to show that $x \in T_\ell$, i.e., x is tainted at ℓ.

Note that the values of public input variables are the same in the initial configurations c_0 and c'_0. We then can deduce that the value of the variable x at the label ℓ depends upon some secret input variables from $\mathbb{X}_h^{in}$. By the soundness of the IFDS framework [64] and sparse static analysis [60], x must be tainted at ℓ. □

Obviously, if for each potential side-channel source (ℓ, x), the variable x is not tainted at the label ℓ, we can deduce that the safe program is constant-time secure.

4.2 Precise Taint Analysis via Taint-directed Semi-cross-product

While the lightweight taint analysis is often effective in ruling out a large number of potential side-channel sources to be genuinely vulnerable, some can still not be determined due to the over-approximation nature of the static analysis, e.g., index-insensitive. In this subsection, we propose a precise taint analysis, which reduces the flow-, context-, path-, field- and index-sensitive taint analysis problem to checking safety properties of a novel cross-product, called *taint-directed semi-cross-product*. We shall first explain the intuition and then present the formal construction.

Intuition. Given a program P, we construct a semi-cross-product P′ of the program P and its Boolean abstraction. Here, 'semi' means that one copy in the cross-product P′ is a Boolean abstraction of P instead of the original one; 'cross' means that P′ shares the same control flow of P and executes statements of two copies in a lockstep manner.

The Boolean abstraction has

- a Boolean variable b_x for each scalar variable x in P such that $b_x = 1$ iff x is tainted.
- a Boolean array b_x for each array variable x in P such that $b_x[i] = 1$ iff $x[i]$ is tainted.

The precise taint analysis is to determine whether a scalar variable x (resp. an array element $x[i]$) is tainted or not. For this purpose, it suffices to check whether there exist inputs to P′ such that b_x (resp. $b_x[i]$) is 1, which is a standard safety verification problem.

To reduce the cost of safety verification, we incorporate the results from the lightweight taint analysis into the semi-cross-product P′ based on the following observation. The lightweight taint analysis is conservative (i.e., intuitively it may overly taint), consequently, a more precise taint analysis would not taint the variables that have not been tainted by the lightweight taint analysis. Hence if a variable x has not been tainted by the lightweight taint analysis, its Boolean abstraction b_x in the semi-cross-product P′ will always be 0. As a result, when verifying P′, it suffices to focus exclusively on the variables that have been tainted by the lightweight taint analysis. For the variables that have not been tainted by the lightweight taint analysis, we can simply assign 0 to them, which can further improve the efficiency of safety verification.

Product construction. The semi-cross-product P′ for a given program P is constructed by iteratively applying the function $\pi(\cdot)$ (given in Fig. 7) to each procedure of P. For each procedure, formal parameters and return variables are duplicated by the function $\pi(\cdot)$ using their Boolean abstractions, and the procedure body p is replaced by $\pi(p)$. Moreover, the Boolean abstractions of public and secret inputs are respectively initialized by 0 and 1 at the beginning of the `main` procedure.

During the construction, $\xi(e)$ is used to generate the Boolean abstraction of the expression e that computes the taint value of the result of the expression e. Specifically, if e is a constant n, $\xi(e)$ is the Boolean constant 0; if e a variable x, $\xi(e)$ is the Boolean abstraction of x (i.e., b_x); and if e is a

$$\xi(n) \triangleq 0 \qquad \xi(x) \triangleq b_x \qquad \xi(e_1 \odot e_2) \triangleq \xi(e_1) \vee \xi(e_2)$$

$$\pi(p) \triangleq p \text{ if } p \text{ is } \texttt{skip} \text{ or } \texttt{assert}\ e \text{ or } \texttt{assume}\ e \qquad \pi(p_1;\ p_2) \triangleq \pi(p_1);\ \pi(p_2)$$

$$\pi(\ell:\ x := e) \triangleq \begin{cases} x := e;\ b_x := 0 & \text{if } \mathsf{var}(e) \cap T_\ell = \emptyset \\ x := e;\ b_x := \xi(e) & \text{otherwise} \end{cases}$$

$$\pi(\ell:\ x := y[z]) \triangleq \begin{cases} \mathsf{Guard}_\ell(z);\ x := y[z];\ b_x := 0 & \text{if } y \notin T_\ell \\ \mathsf{Guard}_\ell(z);\ x := y[z];\ b_x := b_y[z] & \text{otherwise} \end{cases}$$

$$\pi(\ell:\ x[y] := z) \triangleq \begin{cases} \mathsf{Guard}_\ell(y);\ x[y] := z;\ b_x[y] := 0 & \text{if } z \notin T_\ell \\ \mathsf{Guard}_\ell(y);\ x[y] := z;\ b_x[y] := b_z & \text{otherwise} \end{cases}$$

$$\pi(\ell:\ \texttt{while}\ x\ \texttt{do}\ p\ \texttt{od}) \triangleq \texttt{while}\ x@\texttt{INV}\ \texttt{do}\ \mathsf{Guard}_{\mathsf{begin}(\ell)}(x);\ \pi(p)\ \texttt{od};\mathsf{Guard}_{\mathsf{exit}(\ell)}(x)$$

$$\pi(\ell:\ \texttt{if}\ x\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{fi}) \triangleq \mathsf{Guard}_\ell(x);\ \texttt{if}\ x\ \texttt{then}\ \pi(p_1)\ \texttt{else}\ \pi(p_2)\ \texttt{fi}$$

$$\pi(x_1, \cdots, x_m := f(y_1, \cdots, y_n)) \triangleq x_1, b_{x_1}, \cdots, x_m, b_{x_m} := f(y_1, b_{y_1}, \cdots, y_n, b_{y_n})$$

$$\pi(\texttt{def}\ f(x_1, \cdots, x_n)\{p;\ \texttt{return}\ y_1, \cdots, y_m;\}) = \texttt{def}\ f(x_1, b_{x_1} \cdots, x_n, b_{x_n})\{\pi(p);\ \texttt{return}\ y_1, b_{y_1}, \cdots, y_m, b_{y_m};\}$$

$$\mathsf{Guard}_\ell(x) \triangleq (x \in T_\ell\ ?\ \texttt{assert}\ \neg b_x : \texttt{skip})$$

Fig. 7. The taint-directed semi-cross-product of the WHILE programs, where $\mathsf{begin}(\ell)$ and $\mathsf{exit}(\ell)$ denote the labels of the beginning and exit of the loop body of the loop at the label ℓ.

compound expression $e_1 \odot e_2$, $\xi(e)$ is the disjunction $\xi(e_1) \vee \xi(e_2)$ of taint values of sub-expressions, namely, the value of e is tainted if some variable used in e is tainted.

For each statement p with label ℓ, $\pi(p)$ produces a new statement. If p is a `skip` or `assert` or `assume` statement, $\pi(p)$ gives p itself, namely, it does not have a Boolean abstraction. If p is a sequential statement $p_1;\ p_2$, $\pi(p)$ gives the sequential statement $\pi(p_1);\ \pi(p_2)$ by recursively applying the function $\pi(\cdot)$. If p is a standard assignment $x := e$, $\pi(p)$ is defined to track the information flow from the operands of the expression e to the Boolean abstraction b_x of x if some operand of e has been tainted by the lightweight taint analysis, otherwise 0.

If p is a load statement $x := y[z]$, $\pi(p)$ gives a sequential statement $\mathsf{Guard}_\ell(z);\ x := y[z];\ b_x := 0$ if y has not been tainted (i.e., $y \notin T_\ell$). Otherwise, $\mathsf{Guard}_\ell(z);\ x := y[z];\ b_x := b_y[z]$ is generated meaning that b_x is the same as $b_y[z]$. The auxiliary function $\mathsf{Guard}_\ell(z)$ is used to generate an `assert` statement to resolve the taint status of the variable z if z has been tainted, because (ℓ, z) is a potential side-channel source. If z has not been tainted, the assertion is avoided by replacing it with a `skip` statement which can be further removed from the program P'. A store statement $x[y] := z$ is handled similarly, except that $b_y = 0$ is checked by inserting an `assert` statement if y has been tainted (i.e., $y \in T_\ell$) and $b_x[y]$ is updated accordingly.

If p is a `while`-`do` statement `while` x `do` p `od`, $\pi(p)$ inserts an `assert` statement at the beginning (resp. exit) of the loop body to ensure that $b_x = 0$ if x has been tainted at the beginning (resp. exit) of the loop body. Furthermore, to facilitate safety verification, loop invariants INV are inserted. Currently, we use the following heuristic strategy to generate loop invariants and leave the generation of more effective loop invariants as future work. For each variable y defined in the loop body, if it is required to determine some `assert` statement, then the predicate $\neg b_y$ is added as a loop invariant. However, such loop invariants may be invalid in practice. Loop invariants are checked and invalid ones are removed during safety verification.

If p is an `if`-`then`-`else` statement `if` x `then` p_1 `else` p_2 `fi`, similarly, $\pi(p)$ inserts an `assert` statement to resolve the taint status of the branching condition x if it has been tainted.

For every potential side-channel source (ℓ, x) such that $x \in T_\ell$, the program P' must have one or two `assert` statements introduced by `Guard` which checks if $b_x = 0$. By applying a sound safety verifier, we can check if such `assert` statements are valid or not. If they are valid, we can deduce

that (ℓ, x) is not a side-channel source, thus the variable x can be removed from the set of taint facts T_ℓ. We denote by T'_ℓ the resulting set of taint facts. It is trivial to see that Lemma 4.1 still holds when T_ℓ is updated by T'_ℓ for every potential side-channel source (ℓ, x).

Lemma 4.2. *For any potential side-channel source (ℓ, x) and pair (c_0, c'_0) of initial configurations such that $(c_0 \simeq_{\mathbb{X}_l^{in}} c'_0)$, if $x \notin T'_\ell$, then the values of x are the same at the label ℓ in any pair of complete executions $\rho = c_0 c_1 \cdots c_n$ and $\rho' = c'_0 c'_1 \cdots c'_{n'}$.*

Proof sketch. Suppose the values of x are different at the label ℓ in a pair of complete executions $\rho = c_0 c_1 \cdots c_n$ and $\rho' = c'_0 c'_1 \cdots c'_{n'}$ with $c_0 \simeq_{\mathbb{X}_l^{in}} c'_0$. Let b_x be the Boolean abstraction of x. We show that the assert statement `assert` $\neg b_x$ for the potential side-channel source (ℓ, x) is not valid.

As the values of x are different at ℓ in ρ and ρ', and the values of public input variables are the same in the initial configurations c_0 and c'_0, we can deduce that the value of x at ℓ depends upon some secret input variables from $\mathbb{X}_h^{in}$. Since the Boolean abstractions of all the secret input variables are initialized by 1, the Boolean abstraction of each internal variable is set to 0 only if the internal variable is independent of secret input variables, and Boolean abstractions can only be disjunction, we conclude that b_x is 1 when the semi-cross-product P' takes the inputs from c_0 or c'_0. □

4.3 Taint-directed Self-composition

The precise taint analysis can resolve potential side-channel sources that were left by the lightweight taint analysis, but may still fail on some potential side-channel sources. Indeed, it cannot determine genuine side-channel sources that would leak secrets. Thus, in this subsection, we propose a taint-directed self-composition which improves the original construction [11] by incorporating the taint information. Our construction simplifies the self-composed programs and reduces the number of safety checks. We first describe the intuition and then present the formal construction.

Intuition. Given a program P, we construct a cross-product $\hat{\mathsf{P}}$ of the program P comprising two copies of P which share the same control flow. One copy is the *original* program, and the other copy is referred to as the *shadow* one which has a shadow variable $\hat{x}$ for each variable x in P. Typically we check whether the variable x and its shadow $\hat{x}$ have the same values when the original and shadow counterparts of $\hat{\mathsf{P}}$ are provided with the same public inputs but different secret inputs.

To reduce the cost of safety verification, we also incorporate the results from two taint analyses into $\hat{\mathsf{P}}$ based on the following observation. If a variable x has not been tainted, x and its shadow $\hat{x}$ in the product $\hat{\mathsf{P}}$ must have the same values when the public inputs of the two copies are the same. Therefore, when verifying $\hat{\mathsf{P}}$, it suffices to focus exclusively on the variables that are still tainted after two taint analyses. Moreover, for those variables that have not been tainted, we can simply assign the value of the original variable to the shadow one instead of copying the computation, which can further improve the efficiency of safety verification.

Product construction. The cross-product $\hat{\mathsf{P}}$ for a given program P is constructed by iteratively applying the function $\Pi(\cdot)$ (given in Fig. 8) to each procedure of P. Moreover, the shadow counterparts of public inputs are initialized by their original ones at the beginning of the `main` procedure. For each procedure, $\Pi(\cdot)$ is defined similar to the function $\pi(\cdot)$ in semi-cross-product, where the Boolean abstractions are replaced by the shadow counterparts.

More specifically, the auxiliary function $\Xi(e)$ used in defining $\Pi(\ell : x := e)$ now generates a shadow expression that computes the value of expression e over shadow variables. Concretely, if e is a constant n, $\Xi(e)$ is the constant n; if e is a variable x, $\Xi(e)$ is the shadow variable $\hat{x}$ of x; and if e is a compound expression $e_1 \odot e_2$, $\Xi(e)$ is $\Xi(e_1) \odot \Xi(e_2)$.

$$\Xi(n) \triangleq n \qquad \Xi(x) \triangleq \hat{x} \qquad \Xi(e_1 \odot e_2) \triangleq \Xi(e_1) \vee \Xi(e_2)$$

$$\Pi(p) \triangleq p \text{ if } p \text{ is } \texttt{skip} \text{ or } \texttt{assert}\ e \text{ or } \texttt{assume}\ e \qquad \Pi(p_1;\ p_2) \triangleq \Pi(p_1);\ \Pi(p_2)$$

$$\Pi(\ell:\ x := y[z]) \triangleq \begin{cases} \texttt{Guard}'_\ell(z);\ x := y[z];\ \hat{x} := x & \text{if } y \notin T'_\ell \\ \texttt{Guard}'_\ell(z);\ x := y[z];\ \hat{x} := \hat{y}[\hat{z}] & \text{otherwise} \end{cases}$$

$$\Pi(\ell:\ x[y] := z) \triangleq \begin{cases} \texttt{Guard}'_\ell(y);\ x[y] := z;\ \hat{x}[\hat{y}] := x[y] & \text{if } z \notin T'_\ell \\ \texttt{Guard}'_\ell(y);\ x[y] := z;\ \hat{x}[\hat{y}] := \hat{z} & \text{otherwise} \end{cases}$$

$$\Pi(\ell:\ x := e) \triangleq \begin{cases} x := e;\ \hat{x} := x & \text{if } \mathtt{var}(e) \cap T'_\ell = \emptyset \\ x := e;\ \hat{x} := \Xi(e) & \text{otherwise} \end{cases}$$

$$\Pi(\ell:\ \texttt{while}\ x\ \texttt{do}\ p\ \texttt{od}) \triangleq \texttt{while}\ x@\texttt{INV}'\ \texttt{do}\ \texttt{Guard}'_{\texttt{begin}(\ell)}(x);\ \Pi(p)\ \texttt{od};\texttt{Guard}'_{\texttt{exit}(\ell)}(x)$$

$$\Pi(\ell:\ \texttt{if}\ x\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{fi}) \triangleq \texttt{Guard}'_\ell(x);\ \texttt{if}\ x\ \texttt{then}\ \Pi(p_1)\ \texttt{else}\ \Pi(p_2)\ \texttt{fi}$$

$$\Pi(x_1, \cdots, x_m := f(y_1, \cdots, y_n)) \triangleq x_1, \hat{x}_1, \cdots, x_m, \hat{x}_m := f(y_1, \hat{y}_1, \cdots, y_n, \hat{y}_n)$$

$$\Pi(\texttt{def}\ f(x_1, \cdots, x_n)\{p;\ \texttt{return}\ y_1, \cdots, y_m;\}) = \texttt{def}\ f(x_1, \hat{x}_1 \cdots, x_n, \hat{x}_n)\{\Pi(p);\ \texttt{return}\ y_1, \hat{y}_1, \cdots, y_m, \hat{y}_m;\}$$

$$\texttt{Guard}'_\ell(x) \triangleq (x \in T'_\ell\ ?\ \texttt{assert}\ x = \hat{x} : \texttt{skip})$$

Fig. 8. The taint-directed cross-product of the While programs.

The auxiliary function $\texttt{Guard}'_\ell(x)$ for each left potential side-channel source (ℓ, x) generates an `assert` statement to ensure that the variable x and its shadow counterpart have the same value (i.e., $x = \hat{x}$) if x has been tainted. It allows us to resolve the taint status of x.

The duplicated Boolean counterpart $b_x := 0$ (resp. $b_x[y] := 0$) is replaced by $\hat{x} := x$ (resp. $\hat{x}[\hat{y}] := x[y]$) in $\Pi(\ell:\ x := y[z])$ (resp. $\Pi(\ell:\ x[y] := z)$) if x was untainted, otherwise the right-hand side is copied using the corresponding shadow counterparts.

For each $\texttt{while}\ x\ \texttt{do}\ p\ \texttt{od}$, $\Pi(p)$ inserts loop invariants INV. Following [11], for each variable y defined in the loop body, if it is required to determine some `assert` statement, then the predicate $y = \hat{y}$ is added as a loop invariant. Similar to the precise taint analysis, such loop invariants are checked and invalid ones are removed during safety verification.

For every potential side-channel source (ℓ, x) such that $x \in T'_\ell$, the program $\hat{\mathsf{P}}$ must have one or two `assert` statements introduced by $\texttt{Guard}'$ which checks if $x = \hat{x}$. By invoking safety verifiers, we can check if such `assert` statements are valid or not. If they are valid, we can deduce that (ℓ, x) is not a side-channel source, thus the variable x can be removed from the set of taint facts T'_ℓ. We denote by $\hat{T}_\ell$ the resulting set of taint facts. It is trivial to see that Lemma 4.1 still holds when T_ℓ is updated by $\hat{T}_\ell$ for every potential side-channel source (ℓ, x).

Lemma 4.3. *For any potential side-channel source (ℓ, x) and pairs (c_0, c'_0) of initial configurations such that $(c_0 \simeq_{\mathbb{X}_l^{in}} c'_0)$, if $x \notin \hat{T}_\ell$, then the values of x are the same at the label ℓ in any pair of complete executions $\rho = c_0 c_1 \cdots c_n$ and $\rho' = c'_0 c'_1 \cdots c'_{n'}$.*

Proof sketch. The correctness of the lemma follows directly from the correctness of the product construction [11] and Lemmas 4.1—4.2. □

By Lemmas 4.1—4.3, we obtain

Theorem 4.4. *Given a safe program P, if $\hat{T}_\ell = \emptyset$ for any potential side-channel source (ℓ, x), P is constant-time secure.*

5 IMPLEMENTATION AND EVALUATION

We implement our approach in a fully automated prototype tool, named CT-Prover. CT-Prover leverages the static value-flow analysis framework SVF [69] for pre-analysis (i.e., computing call-graph, interprocedural-control-flow graph, points-to information and definition-use chains) and

Phasar [66] for building our lightweight taint analysis. In particular, CT-Prover leverages points-to information from SVF to cope with dynamic memory accesses and the recursive implementation of the IFDS framework in Phasar is rewritten as a worklist-based loop implementation for efficiency consideration. CT-Prover utilizes the SMACK toolchain [63] to translate LLVM IR with taint information into Boogie IR [19] and Bam-Bam-Boogieman [58] to construct product programs on Boogie IR. The Boogie verifier is used as the underlying safety verification engine. Dynamic memory accesses in product programs are handled by SMACK and Boogie.

Research questions. We investigate the following research questions:

RQ1. How effective and efficient is CT-Prover in proving constant-time security?
RQ2. How efficient is CT-Prover for finding side-channel sources?
RQ3. What are the respective contributions of the three main steps in CT-Prover?

Benchmark. We collect 87 real-world examples which are implementations from widely used modern cryptographic and SSL/TLS libraries (e.g., Tongsuo [5], BearSSL [61], Mbed TLS [72], HACL* [2], FourQlib [59], libsodium [3] and OpenSSL [4]) and (constant-time) implementations from fixed-point arithmetic library (e.g., libfixedtimefixedpoint [13], curve25519-donna [1], and MEE-CBC implementations [10]). These benchmarks include cryptographic utilities, arithmetic operations, public and private key cryptography, and algorithms for encryption, decryption, message authentication code (MAC), and digital signature. Among 87 examples, 58 are explicitly claimed to be constant-time by developers.

The statistics of the benchmarks are shown in Table 1, where benchmarks with constant-time claims are marked by†, #Loc shows the number of lines of the analyzed Boogie IR code (similar to LLVM bitcode [11]), and #Src shows the number of all potential side-channel sources. #Loc ranges from 28 to 68,502 (667,518 in total) and #Src ranges from 0 to 7,840 (55,060 in total). Interestingly, we found that 7 (out of 87) benchmarks have no potential side-channel source after being translated into Boogie IR, indicating that they avoid the use of branching and load/store-related statements. We include them because they can be used to validate the tool implementation and measure the verification efficiency, and also because some of them were verified in ct-verif [11].

The experiments were conducted on a machine with Intel Xeon Gold 6342 2.80GHz CPU, 1T RAM, and Ubuntu 20.04.1. All the benchmarks were compiled with `clang-12 -c -emit-llvm -O0 -g -Xclang -disable-O0-optnone` and `opt -mem2reg`, the same as ct-verif [11]. In particular, `O0` disables optimizations to avoid compiler-introduced leakage, `disable-O0-optnone` disables the 'optnone' pass which affects '-mem2reg' and `mem2reg` is used to reduce redundant load/store instructions.

5.1 Verifying Constant-time Implementations (RQ1, RQ3)

To answer RQ1, we verify all 87 programs. CT-Prover returns "*proved*" for 72 benchmarks. (One will see later that the other 15 programs are indeed not constant-time and the constant-time claim made by the developer may be incorrect.) We mainly compare CT-Prover with ct-verif which is the only available tool at the LLVM IR level.

The results are reported in Table 2, where verification time is given in seconds and TO denotes time out (1 hour). ct-verif fails to prove 16 constant-time programs: it runs out of time on 14 programs without outputting any potential side-channel sources (highlighted in red color) and outputs inconclusive verification results on 2 programs within 1 hour (highlighted in blue color). The latter is because of the loop in P7_1 (cf. Example 1) and the type-casting of pointers in P9_4. This indicates that the theoretical completeness of self-composition based approaches may be compromised by the limitation of safety verification. On the other 56 constant-time programs that can be proved by ct-verif and CT-Prover, CT-Prover is about 50 times faster.

Table 1. Statistics of 87 Benchmarks, where **#Loc** and **#Src** show the numbers of lines and potential side-channel sources of the program in Boogie IR, respectively; † indicates that the benchmark is claimed to be constant-time by developers.

Name	Lib/Algorithm/Function	#Loc	#Src
P1_1	OpenSSL_tls1_cbc_remove_padding	1042	71
P1_2	OpenSSL_ssl3_cbc_digest_record	13089	198
P1_3	OpenSSL_ssl3_cbc_remove_padding	351	14
P1_4	OpenSSL_ssl3_cbc_copy_mac	452	15
p2_1†	MAC-then-Encode-then-CBC-Encrypt	22036	396
P3_1†	Hacl_HMAC_compute_sha2_256	27404	1673
P3_2†	Hacl_HMAC_compute_blake2b_32	68502	5289
P3_3†	Hacl_HMAC_legacy_compute_sha1	2149	111
P3_4†	Hacl_HMAC_compute_blake2s_32	56465	4309
P3_5†	Hacl_HMAC_compute_sha2_384	34620	2183
P3_6†	Hacl_HMAC_compute_sha2_512	34565	2177
P3_7†	Hacl_Chacha20_chacha20_decrypt	3303	180
P3_8†	Hacl_Chacha20_chacha20_encrypt	3285	180
P3_9†	Hacl_Curve25519_64_ecdh	1813	70
P3_10†	Hacl_Curve25519_64_scalarmult	1619	67
P3_11†	Hacl_Curve25519_64_secret_to_public	1701	102
P3_12†	Hacl_Poly1305_32_poly1305_mac	2240	121
P3_13†	Hacl_Poly1305_128_poly1305_mac	72	0
P3_14†	Hacl_Poly1305_256_poly1305_mac	72	0
P3_15†	Hacl_Curve25519_51_ecdh	21158	2313
P3_16†	Hacl_Curve25519_51_scalarmult	20964	2310
P3_17†	Hacl_Curve25519_51_secret_to_public	21046	2345
P4_1	Tongsuo_curve448_ossl_x448	6674	446
P4_2	Tongsuo_curve448_derive_pub_key	8994	575
P4_3	Tongsuo_AES_decrypt	3274	1382
P4_4	Tongsuo_AES_encrypt	2982	1126
P4_5	Tongsuo_constant_time_lookup	339	5
P4_6	Tongsuo_curve25519_derive_pub_key	14744	7840
P4_7	Tongsuo_curve25519_ossl_x25519	3561	163
P5_1	Mbed TLS_rsa_decrypt	13797	788
P5_2†	Mbed TLS_mpi_lt_mpi_ct	411	26
P5_3†	Mbed TLS_ct_rsaes_pkcs1_v15_unpadding	629	19
P5_4†	Mbed TLS_mpi_safe_cond_assign	729	43
P5_5†	Mbed TLS_mpi_core_lt_ct	214	5
P5_6†	Mbed TLS_mpi_safe_cond_swap	765	46
P5_7†	Mbed TLS_ct_memcmp	145	7
P5_8†	Mbed TLS_ct_mpi_uint_cond_assign	140	4
P5_9†	Mbed TLS_ct_memcpy_offset	290	5
P5_10†	Mbed TLS_ct_base64_dec_value	309	0
P5_11	Mbed TLS_DES_crypt_cbc	1991	546
P7_1†	libfixedtimefixedpoint_fix_pow_fix_pow	24538	80
P7_2†	libfixedtimefixedpoint_fix_cmp_fix_cmp	591	0
P7_3†	libfixedtimefixedpoint_fix_ln_fix_ln	15200	0
P7_4†	libfixedtimefixedpoint_fix_eq_fix_eq	153	0

Name	Lib/Algorithm/Function	#Loc	#Src
P6_1	BearSSL_AES_small_decrypt	1395	310
P6_2	BearSSL_AES_small_encrypt	945	54
P6_3†	BearSSL_ChaCha20_ct_run	2655	316
P6_4†	BearSSL_GHASH_ctmul32_br_ghash_ctmul	2588	202
P6_5†	BearSSL_RSA_i15_decrypt	5928	229
P6_6†	BearSSL_EC_p256_m15_api_mul	19535	1895
P6_7†	BearSSL_EC_p256_m31_api_mul	9560	796
P6_8†	BearSSL_EC_p256_m64_api_mul	6450	380
P6_9†	BearSSL_GHASH_ctmul64	1060	10
P6_10†	BearSSL_AES_ct_bitslice_encrypt	1345	60
P6_11†	BearSSL_AES_ct_bitslice_decrypt	1788	72
P6_12†	BearSSL_AES_ct_key_sched	2161	100
P6_13	BearSSL_AES_big_cbc_key_schedule	836	290
P6_14	BearSSL_AES_big_cbc_decrypt	1813	565
P6_15	BearSSL_AES_big_cbc_encrypt	1806	565
P6_16†	BearSSL_RSA_i15_pkcs1_sign	6464	264
P6_17	BearSSL_DES_table_cbc_decrypt	1514	555
P6_18	BearSSL_DES_table_cbc_encrypt	1501	555
P6_19†	BearSSL_RSA_i31_pkcs1_sign	5423	235
P6_20	BearSSL_Poly1305_i15_ChaCha20_run	3409	347
P6_21†	BearSSL_Poly1305_ctmul32_ChaCha20_run	4802	425
P6_22†	BearSSL_EC_p256_m62_api_mul	6617	472
P6_23†	BearSSL_GHASH_ctmul_br_ghash_ctmul	1947	125
P6_24†	BearSSL_AES_ct64_bitslice_encrypt	1348	60
P6_25†	BearSSL_AES_ct64_bitslice_decrypt	1791	72
P6_26†	BearSSL_AES_ct64_key_sched	2069	119
P6_27†	BearSSL_RSA_i32_decrypt	4170	179
P6_28†	BearSSL_Poly1305_ctmul_ChaCha20_run	4318	363
P6_29†	BearSSL_Poly1305_ctmulq_ChaCha20_run	6761	415
P6_30†	BearSSL_DES_ct_cbc_decrypt	1670	49
P6_31†	BearSSL_DES_ct_cbc_encrypt	1658	49
P8_1†	FourQlib_ECC_double_eccdouble	28	0
P8_2†	FourQlib_ECC_madd_eccmadd	1632	58
P8_3†	FourQlib_ECC_norm_eccnorm	2390	79
P9_1	libsodium_core_salsa208	1126	9
P9_2	libsodium_aead_chacha20poly1305_decrypt	9311	397
P9_3	libsodium_core_salsa20	1094	9
P9_4	libsodium_stream_chacha20	6620	259
P9_5	libsodium_onetimeauth_poly1305_block	778	25
P9_6	libsodium_hash_sha512_Transformer	29713	2508
P9_7	libsodium_hash_sha256_Transformer	23806	2008
P10_1	curve25519-donna portable implementation	11737	722
P10_2†	curve25519-donna c64 implementation	23936	1599

To partially answer RQ3, we inspect at which step a program can be proved in the verification process of CT-Prover. We found that all these programs (except for P1_1 and P9_4) can be proved using the lightweight taint analysis solely. The lightweight taint analysis cannot determine 4 (resp. 2) potential side-channel sources of P1_1 (resp. P9_4) due to an infeasible branch condition (resp. index-insensitive), which were resolved by the precise taint analysis. While the individual cost of the two taint analyses is lower than ct-verif, the accumulated cost is slightly higher.

Table 2. Results of verifying constant-time implementations, where TO denotes time out (1 hour).

Name	CT-Prover	ct-verif
P1_1	7.55	7.53
P1_2	1.12	26.74
P1_3	0.04	7.25
P1_4	0.05	7.89
p2_1 †	1.44	134.72
P3_1 †	108.80	TO
P3_2 †	177.48	TO
P3_3 †	104.40	TO
P3_4 †	169.71	TO
P3_5 †	109.19	TO
P3_6 †	122.85	TO
P3_7 †	0.12	30.51
P3_8 †	0.13	30.65
P3_9 †	0.10	8.94
P3_10 †	0.09	8.91
P3_11 †	0.09	9.26
P3_12 †	0.09	9.28
P3_13 †	0.04	7.05

Name	CT-Prover	ct-verif
P3_14 †	0.04	6.93
P3_15 †	0.70	TO
P3_16 †	0.67	TO
P3_17 †	0.63	TO
P4_1	0.53	TO
P4_2	0.57	TO
P4_5	0.04	7.42
P4_6	0.35	177.25
P4_7	0.34	14.16
P5_2 †	0.08	7.27
P5_3 †	0.08	7.36
P5_4 †	1.78	7.49
P5_5 †	0.07	7.05
P5_6 †	1.86	7.57
P5_7 †	0.08	6.91
P5_8 †	0.08	6.46
P5_9 †	0.07	6.63
P5_10 †	0.08	7.00

Name	CT-Prover	ct-verif
P6_3 †	0.10	9.25
P6_4 †	0.20	9.04
P6_6 †	23.24	TO
P6_7 †	0.45	TO
P6_8 †	0.21	794.79
P6_9 †	0.05	7.65
P6_10 †	0.07	7.79
P6_11 †	0.08	8.22
P6_12 †	0.07	8.66
P6_20	0.11	12.38
P6_21 †	0.14	14.49
P6_22 †	0.24	977.27
P6_23 †	0.11	8.40
P6_24 †	0.06	7.93
P6_25 †	0.08	8.41
P6_26 †	0.06	8.94
P6_28 †	0.12	12.23
P6_29 †	0.17	13.86

Name	CT-Prover	ct-verif
P6_30 †	0.07	8.26
P6_31 †	0.07	8.51
P7_1 †	0.64	TO
P7_2 †	0.51	7.42
P7_3 †	0.59	21.36
P7_4 †	0.52	7.37
P8_1 †	0.20	6.87
P8_2 †	0.40	21.69
P8_3 †	0.39	37.73
P9_1	0.14	7.86
P9_2	25.69	23.02
P9_3	0.13	7.97
P9_4	16.05	11.91
P9_5	0.06	7.25
P9_6	4.25	44.12
P9_7	3.34	32.98
P10_1	0.55	TO
P10_2 †	10.89	33.18

In summary, CT-Prover can return conclusive results for these benchmarks. Most constant-time implementations can be proved by the lightweight taint analysis solely, whereas the remaining ones can be proved by the precise taint analysis. CT-Prover significantly outperforms the state-of-the-art tool ct-verif for real-world constant-time implementations.

5.2 Verifying Non-constant-time Implementations (RQ2, RQ3)

To answer RQ2, we check the remaining 15 cases which are deemed to be vulnerable. The results are shown in Table 3, where CT-Prover$^-$ refers to CT-Prover *without* the precise taint analysis (i.e., the 2nd main step), a:b in Time (s) respectively denote the execution time after the 2nd and 3rd main step (note: the 1st step is very efficient whose time is negligible), TO denotes time out (6 hours), x:y:z (resp. x:z) in #Src respectively denote the numbers of side-channel sources reported after the 1st, 2nd and 3rd (resp. 1st and 3rd) main step, and ⋆ indicates that some invalid loop invariants are added by ct-verif and thus may miss side-channel sources. We have manually checked all these final potential side-channel sources with their corresponding execution traces produced by CT-Prover, and found that all of them are genuine timing side-channel vulnerabilities (i.e., not false positives). In particular, P6_5, P6_16, P6_19 and P6_27 (from BearSSL) were claimed to be of constant-time.[1]

Both CT-Prover and ct-verif ran out of time on 4 programs (P5_1, P6_5, P6_16 and P6_19). We note that on P5_1 our lightweight taint analysis is still able to find 48 potential side-channel sources based on which we manually confirm that P5_1 is not constant-time. CT-Prover finally found more side-channel sources than ct-verif on 9 (out of 15) programs (marked by ⋆ in Table 3), because ct-verif misses side-channel sources when some additional loop invariants cannot be proved, while CT-Prover still works after automatically removing such loop invariants. On the other programs, CT-Prover and ct-verif report the same side-channel sources. In terms of efficiency, CT-Prover is slightly slower than ct-verif. It is not surprising as CT-Prover involves three main steps while the last two main steps have to perform safety checking. We remark that ct-verif may miss potential side-channel sources since it skips part of the program (e.g., the loop body) when additional loop invariants cannot be proved, which explains its reduced execution time.

[1]We have reported all the potential vulnerabilities in Table 3 to the respective developer(s), and received confirmations for Mbed TLS and BearSSL.

Table 3. Results of verifying non-constant-time implementations, where CT-Prover$^-$ refers to CT-Prover *without* the precise taint analysis (i.e., the 2nd main step), a:b in Time (s) respectively denote the execution time after the 2nd and 3rd main step (note: the 1st step is very efficient whose time is negligible), TO denotes time out (6 hours), x:y:z (resp. x:z) in #Src respectively denote the numbers of side-channel sources reported after the 1st, 2nd and 3rd (resp. 1st and 3rd) main step, and ⋆ indicates that some invalid loop invariants are added by ct-verif and thus may miss side-channel sources.

Name	CT-Prover		ct-verif		CT-Prover$^-$	
	Time (s)	#Src	Time (s)	#Src	Time (s)	#Src
P4_3	24.05:50.84	49:48:48	17.69	32⋆	24.77	49:48
P4_4	21.03:44.76	49:48:48	16.73	32⋆	21.96	49:48
P5_1	TO:TO	48:–:–	TO	–	TO	48:–
P5_11	14.10:30.68	26:16:16	11.81	16	15.20	26:16
P6_1	9.36:19.27	12:1:1	8.46	1	9.13	12:1
P6_2	8.48:17.24	12:1:1	7.45	1	8.46	12:1
P6_5†	983.86:TO	107:71:50	TO	21⋆	TO	107:49
P6_13	8.85:17.87	6:4:4	8.45	4	8.51	6:4
P6_14	13.06:26.57	32:32:32	10.58	0⋆	12.64	32:32
P6_15	13.06:26.35	32:32:32	10.37	0⋆	12.42	32:32
P6_16†	2561.81:TO	107:71:53	TO	21⋆	TO	107:55
P6_17	9.75:20.31	11:8:8	9.29	8	9.48	11:8
P6_18	9.77:20.12	13:8:8	9.26	8	9.63	13:8
P6_19†	3515.77:TO	142:70:39	TO	21⋆	TO	142:46
P6_27†	131.34:503.23	177:45:45	450.57	22⋆	580.50	177:45

To partially answer RQ3, we analyze the respective number of (potential) side-channel sources reported by the three main steps of CT-Prover and two main steps of CT-Prover$^-$ (in the form of x:y:z and x:z in Table 3). We find that the lightweight taint analysis can determine a large number of potential side-channel sources, the precise taint analysis can resolve the remaining few unsolved ones (i.e., P4_3, P4_4, P5_11, P6_1, P6_2, P6_5, P6_13, P6_16, P6_17, P6_18, P6_19, and P6_27), and the left-over ones are often vulnerabilities (i.e., P4_3, P4_4, P5_11, P6_1, P6_2, P6_13, P6_17, P6_18 and P6_27). By comparing with CT-Prover$^-$, we can observe that disabling the precise taint analysis in CT-Prover (i.e., the 2nd main step) may both improve (i.e., P6_5) and degrade (i.e., P6_16 and P6_19) the capability of finding side-channel sources while reducing the verification time. Nevertheless, the precise taint analyses are still useful for finding all the potential side-channel sources when the 3rd main step runs out of time (i.e., P6_5, P6_16, and P6_19), and the first two main steps (i.e., the lightweight and precise taint analysis) are often able to find all the side-channel sources (i.e., P4_3, P4_4, P5_11, P6_1, P6_2, P6_13, P6_14, P6_15, P6_17, P6_18 and P6_27) with comparable or less execution time than CT-Prover$^-$.

```
1 uint32_t br_rsa_i15_private(const br_rsa_private_key *sk){
2     const unsigned char *p = sk->p;   size_t plen = sk->plen;
3   while (plen > 0 && *p == 0) { p++; plen--;} }
```

Fig. 9. Simplified fragment of P6_5 taken from the BearSSL library.

Case study. Fig. 9 shows one side-channel source $(4, *p == 0)$ of P6_5 from BearSSL. Variable `p` points to a buffer storing a large prime (for RSA) which is secret but the loop condition `*p == 0` depends upon the content of the buffer. It leaks the number of leading zero of the large prime.

In summary, CT-Prover is efficient and effective for finding side-channel sources and significantly outperforms ct-verif. Both taint analyses can determine potential side-channel sources, reducing the cost of the subsequent safety verification and manual validation.

Further comparison. We also compare CT-Prover with two sound but incomplete tools Verasco [26] and BINSEC (the latest version of Binsec/Rel [38]). Verasco performs taint analysis by abstract interpretation with bounded loops, while BINSEC uses relational symbolic execution with bounded paths. We use relatively large benchmarks provided by the respective tools with security annotations to reduce the engineering efforts of modifying benchmarks, and compile benchmarks using the constant-time preserving compiler [22].

We observe that CT-Prover is significantly more efficient than Verasco on almost all the benchmarks (80.63 seconds vs. 1,127.63 seconds in total). Moreover, Verasco cannot output traces and values of scalar input variables leading to potential violations, and misses some timing side-channel sources on the DES implementation. Compared with BINSEC, CT-Prover is about 2 times faster than BINSEC when BINSEC's execution time is independent of input size (e.g., curve25519-donna), and the speedup becomes more significant when BINSEC's execution time increases with input size, e.g., BINSEC takes 0.36, 10.28 and 102.48 seconds on the libsodium_chacha20_xor implementation when the input size is 256, 256×512 and 256×5120 bits while CT-Prover takes 11.21 seconds without limiting the input size.

5.3 Threats to Validity

Internal threats. The major internal threat to our evaluation is the correctness of the implementation of CT-Prover which relies on various open-source frameworks and tools. To mitigate this threat, we compared the results of CT-Prover and ct-verif, and manually analyzed all the side-channel sources discovered by both tools to confirm the correctness of CT-Prover. Moreover, we note that these open-source frameworks and tools have been widely used for years so we would have reasonable confidence in their quality. A more scientific way is to build a verified toolchain, which however requires more resources and thus is left as interesting future work.

External threats. The major external threat to our evaluation is the benchmarks, as the performance of CT-Prover may vary with benchmarks. To mitigate this threat, we consider both constant-time and non-constant-time implementations from widely used modern cryptographic and SSL/TLS libraries, as well as benchmarks used in prior work, e.g., [11, 26, 38]. Furthermore, the benchmarks are selected to be diverse in terms of both types and sizes. Another external threat is the translation from source code to LLVM IR which may introduce violations of constant-time, as demonstrated by [38]. To mitigate this threat, benchmarks are only optimized by `-mem2reg` which reduces redundant address-taken variables in LLVM IR.

6 RELATED WORK

Timing side-channels have received considerable attention (cf. [48] for a survey). We roughly classify the current approaches into three categories. The first class does not rely on verification; the second class mostly leverages program analysis; the third class is largely based on verification. There are verification approaches for other side-channels (e.g., [20, 41, 43–47, 62, 84]) and time-balance (e.g., [14, 15, 28]), which are orthogonal to this work.

Approaches based on concrete execution. A large number of approaches have been proposed for detecting and/or quantifying timing side-channel leakages in terms of e.g., channel capacity and Shannon entropy. To this end, for instance, DATA [78], ct-fuzz [51] and CacheQL [83] make use of concrete execution. CANAL [70], Abacus [18], and ENCIDER [82] leverage dynamic symbolic execution, symbolic execution on individual concrete execution traces, and concolic execution. CaType [52] detects timing side-channel vulnerabilities by applying type inference on individual concrete execution traces. In general, this class of approaches are often effective in bug-finding, but cannot prove the absence of timing side-channel leakages.

Approaches based on program analysis. This class of work is more formal which often is able to prove the absence of timing side-channel leakage. CacheAudit [40] bounds timing side-channel leakages by over-approximating side-channel observations using abstract interpretation. CacheS [76] applies abstract interpretation, but its implementation is unsound due to its imprecise treatment of memory. Taint analysis and security type systems have also been applied to detect side-channel leakage by tracking information flow of the secrets [21, 26, 77], varying in accuracy and

efficiency. Moreover, the work [17, 32, 65] bounds the information leakage via symbolic execution and model-counting. While these approaches are often sound, they may raise false positives. Program transformations have been proposed to eliminate potential side-channel sources [7, 35, 79]. Precise detection approaches can reduce the number of potential side-channel sources but the adopted program transformations may bloat the program, making them less efficient to run.

Approaches based on verification. There are mainly two approaches in this class, i.e., self-composition [12] and relational symbolic execution [42]. ct-verif [11] uses a variant of self-composition (i.e., cross-product) to improve the efficiency of safety verification. ct-verif is complete assuming the completeness of the underlying safety checker. Binsec/Rel [36, 38] uses relational symbolic execution enhanced with secret-dependency tracking, untainting and fault-packing to improve the efficiency. Due to the path explosion problem and unsupported dynamic memory allocation, it is only complete up to a given depth of paths, and the size of the symbolic input (keys, plaintext) has to be fixed. ct-verif targets LLVM IR, a target-independent low-level language, while Binsec/Rel targets binary executables. (Note that the compilation from LLVM IR to binary executable may introduce vulnerabilities.)

Our work generally falls into the third category. Similar to ct-verif, we focus on LLVM IR instead of binary executable or source code. Compared with the existing verification approaches, we extensively leverage taint analysis to facilitate self-composition, especially for reducing safety checks and simplifying the self-composed program. A similar idea was adopted in Binsec/Rel, but in a different way. Our new methodology significantly improves both efficiency and effectiveness.

Consideration of micro-architecture. The above approaches did not address micro-architectural features which have also been studied in literature. For instance, Constantine [27] uses dynamic taint analysis; Oo7 [75] uses static taint analysis; Binsec/Haunted [37] uses relational symbolic execution; Pitchfork [34], Spectector [49], SpecuSym [50] and KleeSpectre [74] leverage (dynamic) symbolic execution; Blade [73] uses type system; [80] uses abstract interpretation.

Our work is orthogonal to these investigations, as they usually tackle the vulnerabilities brought by micro-architectural features with the assumption that the program itself is of constant-time.

7 CONCLUSION

In this paper, we have provided practical verification approaches for constant-time implementations of cryptographic libraries. Our methods are based on a novel synergy of taint analysis and safety verification of self-composed programs. We have implemented a cross-platform and fully automated tool CT-Prover working on LLVM IR. The tool has been extensively evaluated on a large set of real-world benchmarks from modern cryptographic and SSL/TLS and fixed-point arithmetic libraries. The experimental results have confirmed the efficacy of our approaches. In particular, compared to the state-of-the-art tool ct-verif, CT-Prover typically demonstrates 2-3 orders of magnitude of improvement, proving more programs and finding new timing leaks.

At the methodology level, we showcase that a combination of lightweight approaches (taint analysis) and heavyweight approaches (self-composition and safety checking) can yield efficiency and completeness. In particular, we demonstrate that self-composition based approaches, which are normally considered to be powerful but costly, can be made scalable with the aid of static analysis.

DATA AVAILABILITY

To foster further research, source code and benchmarks are available at [6].

ACKNOWLEDGEMENT

This work is supported by the Amazon Research Award from Amazon Web Services, National Natural Science Foundation of China under grants No. 62072309 and 61872340, CAS Project for Young Scientists in Basic Research (YSBR-040), ISCAS New Cultivation Project (ISCAS-PYFX-202201), ISCAS Fundamental Research Project (ISCAS-JCZD-202302), Overseas grants from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2022A03, KFKT2023A04), and Birkbeck BEI School Project (EFFECT).

REFERENCES

[1] 2023. *curve25519-donna*. https://github.com/agl/curve25519-donna.
[2] 2023. *HACL**. https://github.com/hacl-star/hacl-star.
[3] 2023. *libsodium*. https://doc.libsodium.org/.
[4] 2023. *OpenSSL*. https://www.openssl.org.
[5] 2023. *Tongsuo*. https://github.com/Tongsuo-Project/Tongsuo.
[6] 2024. *CT-Prover*. https://doi.org/10.5281/zenodo.10683405
[7] Johan Agat. 2000. Transforming Out Timing Leaks. In *Proc. of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 40–53.
[8] Martin R. Albrecht and Kenneth G. Paterson. 2016. Lucky Microseconds: A Timing Attack on Amazon's s2n Implementation of TLS. In *Proc. of the 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 622–643. https://doi.org/10.1007/978-3-662-49890-3_24
[9] Nadhem J. AlFardan and Kenneth G. Paterson. 2013. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *Proc. of the 2013 IEEE Symposium on Security and Privacy*. 526–540. https://doi.org/10.1109/SP.2013.42
[10] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2016. Verifiable side-channel security of cryptographic implementations: constant-time MEE-CBC. In *Proc. of the 23rd International Conference on Fast Software Encryption*. 163–184.
[11] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *Proc. of the 25th USENIX Security Symposium*. 53–70.
[12] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. 2013. Formal verification of side-channel countermeasures using self-composition. *Sci. Comput. Program.* 78, 7 (2013), 796–812. https://doi.org/10.1016/J.SCICO.2011.10.008
[13] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On Subnormal Floating Point and Abnormal Timing. In *Proc. of the IEEE Symposium on Security and Privacy*. 623–639. https://doi.org/10.1109/SP.2015.44
[14] 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 *Proc. of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 362–375. https://doi.org/10.1145/3062341.3062378
[15] Konstantinos Athanasiou, Byron Cook, Michael Emmi, Colm MacCárthaigh, Daniel Schwartz-Narbonne, and Serdar Tasiran. 2018. SideTrail: Verifying Time-Balancing of Cryptosystems. In *Proc. of the 10th International Conference on Verified Software. Theories, Tools, and Experiments*. 215–228.
[16] AWS. 2023. *s2n-tls*. https://github.com/aws/s2n-tls.
[17] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Pasareanu, and Tevfik Bultan. 2016. String analysis for side channels with segmented oracles. In *Proc. of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 193–204. https://doi.org/10.1145/2950290.2950362
[18] Qinkun Bao, Zihao Wang, Xiaoting Li, James R. Larus, and Dinghao Wu. 2021. Abacus: Precise Side-Channel Analysis. In *Proc. of the 43rd IEEE/ACM International Conference on Software Engineering*. 797–809. https://doi.org/10.1109/ICSE43902.2021.00078
[19] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Proc. of the 4th International Symposium on Formal Methods for Components and Objects*. 364–387. https://doi.org/10.1007/11804192_17
[20] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. 2015. Verified Proofs of Higher-Order Masking. In *Proc. of the 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 457–485. https://doi.org/10.1007/978-3-662-46800-5_18
[21] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. 2014. System-level Non-interference for Constant-time Cryptography. In *Proc. of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 1267–1279.

[22] 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

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

[24] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. 2012. The Security Impact of a New Cryptographic Library. In *Proc. of the 2nd International Conference on Cryptology and Information Security in Latin America*, Vol. 7533. 159–176. https://doi.org/10.1007/978-3-642-33481-8_9

[25] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Proc. of the 23rd International Conference on Computer Aided Verification*. 184–190. https://doi.org/10.1007/978-3-642-22110-1_16

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

[27] 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 *Proc. of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 715–733. https://doi.org/10.1145/3460120.3484583

[28] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2018. Symbolic Path Cost Analysis for Side-channel Detection. In *Proc. of the 40th International Conference on Software Engineering: Companion Proceeedings*. 424–425.

[29] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. 2016. T2: Temporal Property Verification. In *Proc. of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Vol. 9636. 387–393. https://doi.org/10.1007/978-3-662-49674-9_22

[30] Billy Bob Brumley and Nicola Tuveri. 2011. Remote Timing Attacks Are Still Practical. In *Proc. of the 16th European Symposium on Research in Computer Security*. 355–371.

[31] David Brumley and Dan Boneh. 2003. Remote Timing Attacks Are Practical. In *Proc. of the 12th USENIX Security Symposium*.

[32] Tevfik Bultan. 2019. Quantifying Information Leakage Using Model Counting Constraint Solvers. In *Proc. of the 11th International Conference on Verified Software: Theories, Tools, and Experiments*, Vol. 12031. 30–35. https://doi.org/10.1007/978-3-030-41600-3_3

[33] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical Foundations for Software Spectre Defenses. In *Proc. of the 43rd IEEE Symposium on Security and Privacy*. 666–680.

[34] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-time foundations for the new spectre era. In *Proc. of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 913–926. https://doi.org/10.1145/3385412.3385970

[35] 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 *Proc. of the 40th ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 174–189. https://doi.org/10.1145/3314221.3314605

[36] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2020. Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level. In *Proc. of the 2020 IEEE Symposium on Security and Privacy*. 1021–1038.

[37] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2021. Hunting the Haunter - Efficient Relational Symbolic Execution for Spectre with Haunted RelSE. In *Proc. of the 28th Annual Network and Distributed System Security Symposium*.

[38] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2023. Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure. *ACM Trans. Priv. Secur.* 26, 2 (2023), 11:1–11:42. https://doi.org/10.1145/3563037

[39] Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, and Frank Piessens. 2023. ProSpeCT: Provably Secure Speculation for the Constant-Time Policy. In *Proc. of the 32nd USENIX Security Symposium*.

[40] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *Proc. of the 22th USENIX Security Symposium*. 431–446.

[41] Yuxin Fan, Fu Song, Taolue Chen, Liangfeng Zhang, and Wanwei Liu. 2022. PoS4MPC: Automated Security Policy Synthesis for Secure Multi-party Computation. In *Proc. of the 34th International Conference on Computer Aided Verification*, Sharon Shoham and Yakir Vizel (Eds.), Vol. 13371. 385–406. https://doi.org/10.1007/978-3-031-13185-1_19

[42] Gian Pietro Farina, Stephen Chong, and Marco Gaboardi. 2019. Relational Symbolic Execution. In *Proc. of the 21st International Symposium on Principles and Practice of Programming Languages*. 10:1–10:14. https://doi.org/10.1145/3354166.3354175

[43] Pengfei Gao, Hongyi Xie, Fu Song, and Taolue Chen. 2021. A Hybrid Approach to Formal Verification of Higher-Order Masked Arithmetic Programs. *ACM Trans. Softw. Eng. Methodol.* 30, 3 (2021), 26:1–26:42. https://doi.org/10.1145/3428015

[44] Pengfei Gao, Hongyi Xie, Pu Sun, Jun Zhang, Fu Song, and Taolue Chen. 2022. Formal Verification of Masking Countermeasures for Arithmetic Programs. *IEEE Trans. Software Eng.* 48, 3 (2022), 973–1000. https://doi.org/10.1109/

TSE.2020.3008852

[45] Pengfei Gao, Hongyi Xie, Jun Zhang, Fu Song, and Taolue Chen. 2019. Quantitative Verification of Masked Arithmetic Programs Against Side-Channel Attacks. In *Proc. of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 155–173. https://doi.org/10.1007/978-3-030-17462-0_9

[46] Pengfei Gao, Jun Zhang, Fu Song, and Chao Wang. 2019. Verifying and Quantifying Side-channel Resistance of Masked Software Implementations. *ACM Trans. Softw. Eng. Methodol.* 28, 3 (2019), 16:1–16:32. https://doi.org/10.1145/3330392

[47] Pengfei Gao, Yedi Zhang, Fu Song, Taolue Chen, and François-Xavier Standaert. 2023. Compositional Verification of Efficient Masking Countermeasures against Side-Channel Attacks. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 1817–1847. https://doi.org/10.1145/3622862

[48] Antoine Geimer, Mathéo Vergnolle, Frédéric Recoules, Lesly-Ann Daniel, Sébastien Bardin, and Clémentine Maurice. 2023. A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries. In *Proc. of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 1690–1704. https://doi.org/10.1145/3576915.3623112

[49] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *Proc. of the IEEE Symposium on Security and Privacy*. 1–19. https://doi.org/10.1109/SP40000.2020.00011

[50] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. 2020. SpecuSym: speculative symbolic execution for cache timing leak detection. In *Proc. of the 42nd International Conference on Software Engineering*. 1235–1247. https://doi.org/10.1145/3377811.3380428

[51] Shaobo He, Michael Emmi, and Gabriela F. Ciocarlie. 2020. ct-fuzz: Fuzzing for Timing Leaks. In *Proc. of the 13th IEEE International Conference on Software Testing, Validation and Verification*. 466–471. https://doi.org/10.1109/ICST46399.2020.00063

[52] Ke Jiang, Yuyan Bao, Shuai Wang, Zhibo Liu, and Tianwei Zhang. 2022. Cache Refinement Type for Side-Channel Detection of Cryptographic Software. In *Proc. of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1583–1597. https://doi.org/10.1145/3548606.3560672

[53] 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. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proc. of IEEE Symposium on Security and Privacy*. 1–19.

[54] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proc. of the 16th Annual International Cryptology Conference on Advances in Cryptology*. 104–113.

[55] Haofeng Li, Haining Meng, Hengjie Zheng, Liqing Cao, Jie Lu, Lian Li, and Lin Gao. 2021. Scaling up the IFDS algorithm with efficient disk-assisted computing. In *Proc. of the IEEE/ACM International Symposium on Code Generation and Optimization*. 236–247.

[56] Peng Li, Debin Gao, and Michael K. Reiter. 2014. StopWatch: A Cloud Architecture for Timing Channel Mitigation. *ACM Trans. Inf. Syst. Secur.* 17, 2 (2014), 8:1–8:28.

[57] 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. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proc. of the 27th USENIX Security Symposium*. 973–990.

[58] Michael Emmi. 2023. *bam-bam-boogieman*. https://github.com/michael-emmi/bam-bam-boogieman.

[59] Microsoft. 2023. *FourQlib*. https://github.com/microsoft/FourQlib.

[60] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. Design and implementation of sparse global analyses for C-like languages. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 229–238.

[61] Thomas Pornin. 2023. *BearSSL*. https://bearssl.org.

[62] Qi Qin, JulianAndres JiYang, Fu Song, Taolue Chen, and Xinyu Xing. 2022. DeJITLeak: eliminating JIT-induced timing side-channel leaks. In *Proc. of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 872–884. https://doi.org/10.1145/3540250.3549150

[63] Zvonimir Rakamarić and Michael Emmi. 2014. SMACK: Decoupling source language details from verifier implementations. In *Proc. of the 26th International Conference on Computer Aided Verification*. 106–113. https://doi.org/10.1007/978-3-319-08867-9_7

[64] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proc. of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 49–61.

[65] Seemanta Saha, Surendra Ghentiyala, Shihua Lu, Lucas Bang, and Tevfik Bultan. 2023. Obtaining Information Leakage Bounds via Approximate Model Counting. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1488–1509. https://doi.org/10.1145/3591281

[66] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In *Proc. of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 393–410. https://doi.org/10.1007/978-3-030-17465-1_22
[67] Amazon Web Services. 2023. *ct-verif for s2n*. https://github.com/aws/s2n-tls/tree/main/tests/ctverif.
[68] Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *Proc. of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 57–69.
[69] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proc. of the 25th international conference on compiler construction*. 265–266. https://doi.org/10.1145/2892208.2892235
[70] Chungha Sung, Brandon Paulsen, and Chao Wang. 2018. CANAL: a cache timing analysis framework via LLVM transformation. In *Proc. of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 904–907. https://doi.org/10.1145/3238147.3240485
[71] Tachio Terauchi and Alexander Aiken. 2005. Secure Information Flow as a Safety Problem. In *Proc. of the 12th International Symposium on Static Analysis*. 352–367. https://doi.org/10.1007/11547662_24
[72] Trusted Firmware Project. 2023. *Mbed TLS*. https://github.com/Mbed-TLS/mbedtls.
[73] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean M. Tullsen, and Deian Stefan. 2021. Automatically eliminating speculative leaks from cryptographic code with blade. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434330
[74] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. 2020. KLEESpectre: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution. *ACM Trans. Softw. Eng. Methodol.* 29, 3 (2020), 14:1–14:31. https://doi.org/10.1145/3385897
[75] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2021. oo7: Low-Overhead Defense Against Spectre Attacks via Program Analysis. *IEEE Trans. Software Eng.* 47, 11 (2021), 2504–2519. https://doi.org/10.1109/TSE.2019.2953709
[76] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. 2019. Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation. In *Proc. of the 28th USENIX Security Symposium*. 657–674.
[77] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. 2019. CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem. *Proc. of the ACM on Programming Languages* 3, POPL (2019), 77:1–77:29. https://doi.org/10.1145/3290390
[78] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. 2018. DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In *Proc. of the 27th USENIX Security Symposium*. 603–620.
[79] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating Timing Side-Channel Leaks using Program Repair. In *Proc. of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 15–26. https://doi.org/10.1145/3213846.3213851
[80] Meng Wu and Chao Wang. 2019. Abstract interpretation under speculative execution. In *Proc. of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 802–815. https://doi.org/10.1145/3314221.3314647
[81] Weikun Yang, Yakir Vizel, Pramod Subramanyan, Aarti Gupta, and Sharad Malik. 2018. Lazy Self-composition for Security Verification. In *Proc. of the 30th International Conference on Computer Aided Verification*. 136–156. https://doi.org/10.1007/978-3-319-96142-2_11
[82] Tuba Yavuz, Farhaan Fowze, Grant Hernandez, Ken Yihang Bai, Kevin R. B. Butler, and Dave Jing Tian. 2023. ENCIDER: Detecting Timing and Cache Side Channels in SGX Enclaves and Cryptographic APIs. *IEEE Trans. Dependable Secur. Comput.* 20, 2 (2023), 1577–1595. https://doi.org/10.1109/TDSC.2022.3160346
[83] Yuanyuan Yuan, Zhibo Liu, and Shuai Wang. 2023. CacheQL: Quantifying and Localizing Cache Side-Channel Vulnerabilities in Production Software. In *Proc. of the 32nd USENIX Security Symposium*.
[84] Jun Zhang, Pengfei Gao, Fu Song, and Chao Wang. 2018. SCInfer: Refinement-Based Verification of Software Countermeasures Against Side-Channel Attacks. In *Proc. of the 30th International Conference on Computer Aided Verification*. 157–177. https://doi.org/10.1007/978-3-319-96142-2_12

Received 2023-09-28; accepted 2024-01-23