Verifying Constant-Time Implementations

José Bacelar Almeida
HASLab – INESC TEC and University of Minho

Manuel Barbosa
HASLab – INESC TEC and University of Porto

Gilles Barthe
François Dupressoir
Michael Emmi
IMDEA Software Institute

Abstract

A key threat to the secure exchange of digital information is software timing attacks, wherein an attacker learns private information by measuring quantities like program execution time or memory access patterns. "Constant time" implementations of security-critical algorithms are popular defenses against timing attacks. A constant-time implementation has the property that sensitive quantities like program execution time vary only as a function of publicly-disclosed values like encrypted messages, and remain constant with respect to secret information like private encryption keys. Since variations in runtime quantities like program execution time are not always evident to programmers at the source level, verifying that their implementations are indeed constant time is thus crucial to building secure software.

In this work we develop an approach to verifying whether software implementations are constant time. Our approach is parameterized over attacker models, and works by reducing the security of a given program with respect to an attacker model to the safety of the same program instrumented with additional variables and assertions, enabling the use of off-the-shelf safety-verification tools. We use our prototype implementation of this technique to verify the actual C-code implementations of several constant-time algorithms, many of which could not be handled by prior techniques.

1. Introduction

Cryptography lies at the heart of digital security infrastructure. Cryptographic software, however, can only fulfill its role as the backbone of security architectures if it is defect-free. Rigorous security proofs that follow the paradigm of provable security considerably reduce the eventuality that the design of a cryptographic protocol is flawed. Unfortunately, cryptographic software implementations are still far from perfect, as witnessed by recent vulnerabilities in the pervasively-used OpenSSL library, which include implementation bugs leading to buffer overflow attacks, e.g., HeartBleed, and physical side-channel attacks based on measuring execution time, e.g., Lucky 13.

This gap between theoretical security and real-world security is a recognized problem in the field of cryptography. The root cause for this gap is abstraction: the soundness of cryptographic designs is proven in an idealized execution model where the algorithmic descriptions elide many potentially-critical details; these details must be filled by implementors, who may not possess the required theoretical expertise to make sound decisions. Furthermore, attackers targeting real-world platforms may exploit leakage via physical side-channels to break a system, which is absent in the black-box abstractions in which proofs are obtained.

Considering low-level, rather than source, implementations is crucial to minimize exposure to side-channel attacks, because meaningful models of leakage may not exist for source languages, and because optimizing compilers generally do not preserve leakage. As a consequence of this, cryptography implementors face a tough challenge: given a high-level specification of a cryptographic protocol, produce efficient and functionally-correct implementations in low level languages, whilst enforcing non-functional secure-coding policies that mitigate side-channel attacks.

In this work we focus on timing attacks, a particular class of side-channel attacks that are launched by extracting secret information from patterns that can be detected in the execution time of cryptographic implementations. Timing attacks can be launched by direct measurement of the execution time of an implementation in the host platform [17], remote measurement by interacting with an implementation via a network [11], and even by inferring memory-access patterns via the access-time correlation that cache-sharing creates between two processes hosted in the same machine [8, 26, 28].

Software implementations are more vulnerable to timing attacks than hardware implementations, since the practical effectiveness of the attack depends heavily on the variability of the timing behavior, which is typically very small in cryptographic hardware implementations. Conversely, general-purpose microprocessors are designed to run code efficiently, but ensuring that execution time does not depend on possibly-secret data is *not* a design criteria. Careless software implementations of cryptographic components can therefore open the door to complete security breaches, for example by inferring long-term secret keys by determining which of two unbalanced control paths were executed [8, 11], or whether certain memory accesses lead to cache misses [17, 28].

1.1 Constant-Time Implementations

To mitigate timing attacks, cryptography practitioners often enforce strict secure coding policies that give rise to so-called *constant-time implementations*. The desired property of these implementations is that secret information does not leak into the sequence of memory accesses carried out during execution, both for code and data memory. The underlying adversarial model, which today is referred to as the constant-time model, conservatively considers the worst-case scenario in which an attacker gets a full trace of addresses accessed in code memory (i.e., the instruction sequence) and a full trace of addresses accessed in data memory [9].

Constant-time implementations are sometimes criticized for implying a large performance penalty (despite recent evidence that contradicts this argument [16]) and also for forcing developers to deviate from consolidated software development practices. Indeed, constructing constant time implementations is difficult, and it implies both the adoption of unconventional coding practices and implementation in low-level programming languages. As an example, a constant time implementation must eliminate all branching instructions that could depend on secret information, and so these are replaced by arithmetic constructions, e.g., using

`v = (b)*v_1 + (1-b)*v_0` to perform a guarded assignment. However, it is consensual among practitioners that constant time implementations are the best software-based countermeasure against side channel attacks and, indeed, many cryptography and security libraries include constant time code in their implementations.

This raises the question of how to validate constant-time implementations. The recent case of the S2N library [30] released by Amazon Web Services - Labs (AWS-Labs) shows that the deployment of timing countermeasures (even those more permissive than strict constant-time) is extremely hard to validate. Indeed, soon after the library was released, two patches[1] were applied to the sources in order to protect it from timing attacks, the second of which to correct a timing vulnerability introduced by the first patch itself! These problems were reported despite the code having been subject to at least three code reviews and extensive testing procedures. The natural conclusion is that standard software validation processes are not adequate for this purpose, and new techniques must be developed.

1.2 Verifying Constant Time

In this work we develop an approach to verifying whether software implementations that are intended to be constant time are actually constant time. In order to provide flexibility across a wide range of machine architectures and attacker models, we parameterize our approach by abstract notions of machine states and observations, along with a function mapping machine states to observations. Our parameterization captures many conceivable timing attacks including program execution time and memory access patterns.

Given an attacker model, our technique works by instrumenting a given program with additional data and assertions, and verifying the absence of assertion violations in the instrumented program. We thus reduce the verification of security properties, which are properties over pairs of program executions, to the verification of safety properties, over single executions. Technically, we achieve this reduction by constructing the instrumented program to simulate pairs of executions along the same program-control paths, with separate copies of program data. This reduction is linear time: the size of the instrumented program is within a small constant factor of the original. Our reduction-based approach is easily implementable by code-to-code translation at the compiler IR level. Performing the reduction at this level is ideal due to the availability of analyses and tools, and the similarity to the machine code level, yielding relatively-high fidelity between verification results and actual machine behavior.

Based on this reduction we develop a prototype tool which is capable of verifying the actual C-code implementations of many constant-time algorithms, many of which could not be verified by prior techniques due to expressivity limitations or scalability bottlenecks. Our tool shows that strict constant time coding policies have an additional advantage that has so far gone unnoticed: they are susceptible to automatic formal verification. We demonstrate this fact by presenting an extensive set of examples of off-the-shelf cryptographic libraries, which we have been able to verify with minimal verification effort.

1.3 Contributions

This work makes three fundamental contributions:

- a linear-time reduction from the verification of security properties to the verification of safety properties, parameterized by an expressive attacker model,
- an end-to-end implementation of our reduction-based approach which is capable of verifying the actual C-code implementations of many constant-time algorithms, and
- a case study involving the verification of many constant-time algorithms, several for which verification was beyond the capabilities of prior approaches.

Our implementation and case studies are publicly available[2] and have led both to the discovery of unknown timing leaks and the validation of novel fixes to known timing leaks.

1.4 Outline

We begin in Section 2 by developing a notion of security parameterized over attacker models, and Section 3 describes a reduction from the security of a program to the safety of the same program instrumented with additional variables and assertions. Section 4 describes our implementation of a verifier of constant time for C code using this reduction, and in Section 5 we discuss our extensive case study in verifying actual C-code implementations. We discuss related work in Section 6.

2. Security Against Timing Attacks

In this section we formalize a notion of security against various timing attacks. In order to describe our verification approach in a generic setting, independently of the computing platform and the nature of the exploits, we introduce abstract notions of *machines* and *observations*.

2.1 An Abstract Computing Platform

Formally, a *machine* $M = \langle A, B, V, I, O, F, T\rangle$ consists of

- a set A of address-space names,
- a set B of control-block names — these names determine the set $K = \{\texttt{fail}, \texttt{halt}, \texttt{spin}, \texttt{next}, \texttt{jump}(b) : b \in B\}$ of control codes,
- a set V of values — each address space $a \in A$ corresponds to a subset $V_a \subset V$ of values; together, the address spaces and values determine the set $S = A \to (V_a \to V)$ of states, each $s \in S$ mapping $a \in A$ to value store $s_a : V_a \to V$; we write $a{:}v$ to denote a reference to $s_a(v)$,
- a set I of instructions — operands are references $a{:}v$, block names $b \in B$, and literal values,
- a set O of observations, including the null observation ε,
- a function $F : S \times I \to O$ determining the observation at each state-instruction pair, and
- a transition function $T : S \times I \to S \times K$ from states and instructions to states and control codes.

We assume that the value set V includes the integer value $0 \in \mathbb{N}$, that the control-block names include `entry`, and that the instruction set I includes the following:

$$T(s, \texttt{assume}\ a{:}v) = \begin{cases} \langle s, \texttt{spin}\rangle & \text{if } s_a(v) = 0 \\ \langle s, \texttt{next}\rangle & \text{otherwise} \end{cases}$$

$$T(s, \texttt{assert}\ a{:}v) = \begin{cases} \langle s, \texttt{fail}\rangle & \text{if } s_a(v) = 0 \\ \langle s, \texttt{next}\rangle & \text{otherwise} \end{cases}$$

$$T(s, \texttt{goto}\ b) = \langle s, \texttt{jump}(b)\rangle$$

$$T(s, \texttt{halt}) = \langle s, \texttt{halt}\rangle$$

We write $s[a{:}v_1 \mapsto v_2]$ to denote the state s' identical to s except that $s'_a(v_1) = v_2$.

Example 1. *Consider a machine with register and memory address spaces* r *and* m*, and instructions for memory access, arithmetic,*

[1] See merge pull requests #147 and #179 in the GitHub repository for S2n at `github.com/awslabs/s2n`.

[2] LINK TO REPOSITORY REMOVED FOR DOUBLE BLIND REVIEW

```
extern int u;
extern int v;
extern int w;
int main(void) {
  ...
  if (u) {
    assert(v > 0);
    w = u + v;
  } else {
    v = w;
  }
  return 0;
}
```

```
entry: goto b0
b0:    load[a] r:u r:x
       branch r:x b1
       goto b2
b1:    load[a] r:v r:y
       assert r:y
       add r:x r:y r:z
       store[m] r:w r:z
       goto b3
b2:    load[m] r:w r:z
       store[m] r:v r:z
       goto b3
b3:    store[m] r:r 0
       goto exit
exit:  halt
```

Figure 1. Simple C programming language code along with the corresponding program for the machine of Example 1.

and branching:

$$T(s, \texttt{load}[a]\ \mathrm{r}{:}x\ \mathrm{r}{:}y) = \langle s[\mathrm{r}{:}y \mapsto s_a(s_{\mathrm{r}}(x))], \texttt{next}\rangle$$
$$T(s, \texttt{store}[a]\ \mathrm{r}{:}x\ \mathrm{r}{:}y) = \langle s[a{:}s_{\mathrm{r}}(x) \mapsto s_{\mathrm{r}}(y)], \texttt{next}\rangle$$
$$T(s, \texttt{add}\ \mathrm{r}{:}x\ \mathrm{r}{:}y\ \mathrm{r}{:}z) = \langle s[\mathrm{r}{:}z \mapsto s_{\mathrm{r}}(x) + s_{\mathrm{r}}(y)], \texttt{next}\rangle$$
$$T(s, \texttt{branch}\ \mathrm{r}{:}x\ b) = \begin{cases} \langle s, \texttt{next}\rangle & \text{if } s_{\mathrm{r}}(x) = 0 \\ \langle s, \texttt{jump}(b)\rangle & \text{otherwise} \end{cases}$$

where we parameterize memory-access operations by an address-space name $a \in A$. For the time being we suppose all accesses use the memory address space m*; later, in Example 4 we refine this notion. To model the information exploited in typical timing attacks, we consider the observations*

$$F(s, \texttt{load}[a]\ \mathrm{r}{:}x\ \_) = s_{\mathrm{r}}(x)$$
$$F(s, \texttt{store}[a]\ \mathrm{r}{:}x\ \_) = s_{\mathrm{r}}(x)$$
$$F(s, \texttt{add}\ \_\ \_\ \_) = \varepsilon$$
$$F(s, \texttt{branch}\ \mathrm{r}{:}x\ \_) = s_{\mathrm{r}}(x)$$

to capture variations in memory accesses and branching.

Programs are essentially blocks containing instructions. Formally, a *location* $\ell = \langle b, n\rangle$ is a block name $b \in B$ and index $n \in \mathbb{N}$; the location $\langle \texttt{entry}, 0\rangle$ is called the *entry location*, and L denotes the set of all locations. The location $\langle b, n\rangle$ is the *next successor* of $\langle b, n-1\rangle$ when $n > 0$, and is the *start of block* b when $n = 0$. A *program for machine* M is a function $P : L \to I$ labeling locations with instructions.

Example 2. *Figure 1 lists C programming language code along with the corresponding program for the machine of Example 1. For the moment, we suppose all memory accesses use the memory address space, i.e., $a =$* m*. The program accesses three memory cells via pointers* r:u*,* r:v*, and* r:w*. We write* $*$r:u *to denote the value stored in the memory cell pointed to by* r:u*, like the dereferencing operation in C. The* `entry` *block jumps to block b_1 unless* $*$r:$u = 0$*, in which case control moves to block b_2. Block b_1 stores the sum of* $*$r:u *and* $*$r:v *into the memory cell pointed to by* r:w*. Block b_2, instead, stores* $*$r:w *into the memory cell pointed to by* r:v*.*

2.2 Executions and Traces

A *configuration* $c = \langle s, \ell\rangle$ *of machine* M consists of a state $s \in S$ along with a location $\ell \in L$, and is called

- *initial* when ℓ is the entry location,
- *failing* when $T(s, P(\ell)) = \langle \_, \texttt{fail}\rangle$,
- *halting* when $T(s, P(\ell)) = \langle \_, \texttt{halt}\rangle$, and
- *spinning* when $T(s, P(\ell)) = \langle \_, \texttt{spin}\rangle$.

The *observation* at c is $F(s, P(\ell))$, and configuration $\langle s_2, \ell_2\rangle$ is the *successor* of $\langle s_1, \ell_1\rangle$ when $T(s_1, P(\ell_1)) = \langle s_2, k\rangle$ and

- $k = \texttt{next}$ and ℓ_2 is the next successor of ℓ_1,
- $k = \texttt{jump}(b_2)$ and ℓ_2 is the start of block b_2, or
- $k = \texttt{spin}$ and $\ell_2 = \ell_1$.

We write $c_1 \to c_2$ when c_2 is the successor of c_1, and C denotes the set of configurations.

An *execution of program P for machine M* is a configuration sequence $e = c_0 c_1 \ldots \in (C^* \cup C^\omega)$ such that $c_{i-1} \to c_i$ for each $0 < i < |e|$, and $c_{|e|-1}$ is failing or halting if $|e|$ is finite, in which case we say that e is failing or halting, respectively. The *trace* of e is the sequence $o_0 o_1 \ldots$ of observations of at $c_0 c_1 \ldots$ concatenated, where $o \cdot \varepsilon = \varepsilon \cdot o = o$. Executions with the same trace are *indistinguishable*.

Example 3. *Consider the program from Example 2, and suppose that the registers* r:u*,* r:v*,* r:w *and* r:r *store distinct values u_0, v_0, w_0 and r_0 initially, and that registers* r:x *and* r:y *are loaded with values x_0 and y_0. There are three possible execution traces:*

$u_0\ x_0\ w_0\ v_0\ r_0$	*when* $x_0 = 0$	*(along b_2)*
$u_0\ x_0\ v_0$	*when* $x_0 \neq 0$ *and* $y_0 = 0$	*(along b_1)*
$u_0\ x_0\ v_0\ w_0\ r_0$	*when* $x_0 \neq 0$ *and* $y_0 \neq 0$	*(along b_1)*

for fixed values u_0, v_0, w_0, x_0, y_0, and r_0. The first and third traces arise from halting executions, and the second from a failing execution.

Definition 1 (Safety)**.** *A program P on machine M is* safe *when no executions fail. Otherwise, P is* unsafe*.*

2.3 Security Properties

To define our security property we must relate program traces to input and output values, since, generally speaking, the observations made along executions should very well depend on, e.g., publicly-known input values. Security thus relies on distinguishing program inputs and outputs as public or private. We make this distinction formally using address spaces, supposing that machines include

- a public input address space $\mathrm{i} \in A$,
- a declassified output address space $\mathrm{d} \in A$, and
- a public output address space $\mathrm{o} \in A$,

in addition to, e.g., register and memory address spaces of Example 1. A program which does not use the declassified output address space d is called *declassification-free*.

Intuitively, the observations made on a machine running a secure program should depend only on the initial machine state in the public input address space; when observations depend on non-public inputs, leakage occurs. Similarly, public outputs should depend only on public inputs. A subtlety arises for programs which "declassify" information during execution: observations, and public outputs, can ultimately depend on not only public inputs, but also declassified outputs. Formally, we say that that two states $s_1, s_2 \in S$ are *a-equivalent* for $a \in A$ when $s_1(a)(v) = s_2(a)(v)$ for all $v \in V_a$. Executions e_1 from state s_1 and e_2 from state s_2 are *initially a-equivalent* when s_1 and s_2 are a-equivalent, and finite executions to s_1' and s_2' are *finally a-equivalent* when s_1' and s_2' are a-equivalent.

Definition 2 (Security)**.** *A program is* secure *when:*

1. *Initially* i*-equivalent and finally* d*-equivalent executions are finally* o*-equivalent and indistinguishable.*
2. *Initially* i*-equivalent infinite executions are indistinguishable.*

Otherwise, P is insecure.

The absence of declassification simplifies this definition, since the executions of declassification-free programs are finally d-equivalent, trivially.

Example 4. *The program of Figure 1 is insecure when $a = \mathrm{m}$ since the initial load of $\mathrm{r}{:}u$ uses the memory address space, and there are initially* i*-equivalent executions with distinct traces: e.g., the first and third shown in Example 3. However, if the loads of $\mathrm{r}{:}u$ and $\mathrm{r}{:}v$ use the input address space, i.e., $a = \mathrm{i}$, then the values x_0 and y_0 loaded into registers $\mathrm{r}{:}x$ and $\mathrm{r}{:}y$ are guaranteed to be identical in* i*-equivalent executions, and thus* i*-equivalent executions have identical traces. Thus the program of Example 2 is secure if and only if $a = \mathrm{i}$. This distinction between public and private inputs is typically made in program source code, e.g., by annotating program variables and/or memory locations.*

3. Reducing Security to Safety

According to standard notions like that given in Section 2, security is a property over pairs of executions: a program is secure so long as executions with the same public inputs and declassified outputs are indistinguishable. In this section we demonstrate a reduction from security to safety, which is a property over single executions. The reduction works by instrumenting the original program with additional instructions which simulate two executions of the same program side-by-side, along the same control locations, over two separate address spaces: the original, along with a *shadow* of the machine state. In order for our reduction to be sound, i.e., to witness all security violations as safety violations, the control paths of the simulated executions must not diverge unless they yield distinct observations — in which case our reduction yields a safety violation. This soundness requirement can be stated via the following machine property.

Definition 3 (Control Transparence)**.** *A machine M is* control transparent *when for all states $s \in S$ and instructions $i_1, i_2 \in I$, then the transitions $T(s, i_1) = \langle \_, k_1 \rangle$ and $T(s, i_2) = \langle \_, k_2 \rangle$ yield the same control codes $k_1 = k_2$ whenever the observations $F(s, i_1) = F(s, i_2)$ are identical.*

For the remainder of this work, we suppose that machines are control transparent. Besides control transparence, our construction also makes the following modest assumptions:

- address spaces can be separated and renamed, and
- observations are accessible via instructions.

We capture the first requirement by assuming that programs use a limited set $A_1 \subset A$ of the possible address-space names, and fixing a function $\alpha : A_1 \to A_2$ whose range $A_2 \subset A$ is disjoint from A_1. We then lift this function from address-space names to instructions, i.e., $\alpha : I \to I$, by replacing each reference $a{:}v$ with $\alpha(a){:}v$. We capture the second requirement by assuming the existence of a function $\beta : I \times A \times V \to I$ such that

$$T(s, \beta(i, a, v)) = \langle s[a{:}v \mapsto F(s, i)], \texttt{next} \rangle.$$

For a given instruction $i \in I$, address space $a \in A$, and value $v \in V$, the instruction $\beta(i, a, v)$ stores the observation $F(s, i)$ in state $s \in S$ at $a{:}v$.

We present our reduction for declassification-free programs first in Section 3.1, which we then extend in Section 3.2 to handle programs with output declassification.

3.1 Reduction for Declassification-Free Programs

We suppose machines include a vector-equality instruction

$$T(s, \texttt{eq}\ a_x{:}\vec{x}\ a_y{:}\vec{y}\ a{:}z) = \langle s[a{:}z \mapsto v], \texttt{next} \rangle$$

where $\vec{x}$ and $\vec{y}$ are equal-length vectors of values, and $v = 0$ iff $s(a_x)(x_n) \neq s(a_y)(y_n)$ for some $0 \leq n < |\vec{x}|$. This requirement is for convenience only; technically only a simple scalar-equality instruction is necessary.

To facilitate the checking of initial/final range equivalences for security annotations we assume that a given program P has only a single `halt` instruction, and

$$P(\texttt{entry}, 0) = \texttt{goto}\ b_0$$
$$P(\texttt{exit}, 0) = \texttt{halt}.$$

This is without loss of generality since any program can easily be rewritten in this form.

Given the above functions α and β, and a fresh address space a, the *shadow product* of a program P is the program $P_\times$ defined by an entry block which spins unless the public input values in both i and $\alpha(\mathrm{i})$ address spaces are equal,

$$P_\times(\texttt{entry}, n) = \begin{cases} \texttt{eq}\ \mathrm{i}{:}V_\mathrm{i}\ \alpha(\mathrm{i}){:}V_\mathrm{i}\ a{:}x & n = 0 \\ \texttt{assume}\ a{:}x & n = 1 \\ \texttt{goto}\ b_0 & n > 1 \end{cases}$$

an exit block which fails unless the public output values in both o and $\alpha(\mathrm{o})$ address spaces are equal,

$$P_\times(\texttt{exit}, n) = \begin{cases} \texttt{eq}\ \mathrm{o}{:}V_\mathrm{o}\ \alpha(\mathrm{o}){:}V_\mathrm{o}\ a{:}x & n = 1 \\ \texttt{assert}\ a{:}x & n = 2 \\ \texttt{halt} & n > 2 \end{cases}$$

and finally a rewriting of every other block $b \notin \{\texttt{entry}, \texttt{exit}\}$ of P to run each instruction on two separate address spaces,

$$P_\times(b, n) = \begin{cases} \beta(i, a, x) & n = 0 \pmod 6 \\ \beta(\alpha(i), a, y) & n = 1 \pmod 6 \\ \texttt{eq}\ a{:}x\ a{:}y\ a{:}z & n = 2 \pmod 6 \\ \texttt{assert}\ a{:}z & n = 3 \pmod 6 \\ i & n = 4 \pmod 6 \\ \alpha(i) & n = 5 \pmod 6 \end{cases}$$
$$\text{where } i = P(b, n/6)$$

while asserting that the observations of each instruction are the same along both simulations.

Theorem 1. *P is safe and secure iff $P_\times$ is safe.*

Note that $P_\times$ is unsafe if P is, whether or not P is secure.

Example 5. *Figure 2 lists the shadow product for the program of Figure 1. Note the use of* `cp` *and* `zero` *instructions,*

$$T(s, \texttt{cp}\ a{:}x\ a_y{:}y) = \langle s[a_y{:}y \mapsto s_a(x)], \texttt{next} \rangle$$
$$T(s, \texttt{zero}\ a{:}x) = \langle s[a{:}x \mapsto 0], \texttt{next} \rangle$$

to capture the observations of memory and branch instructions by copying their first argument, and the null observations of `goto` *instructions, respectively. Also note that this program is safe iff the loads of $\mathrm{r}{:}u$ and $\mathrm{r}{:}v$ use the input address space, i.e., $a_0 = \mathrm{i}$, in which case $*\mathrm{r}{:}u = *\alpha(\mathrm{r}){:}u$ and $*\mathrm{r}{:}v = *\alpha(\mathrm{r}){:}v$ is assumed upon executing the* `entry` *block.*

3.2 Reduction for Programs with Declassification

The control paths of initially i-equivalent executions of secure programs may diverge in the presence of declassification. Since the simple shadow-product construction of Section 3.1 assumes that control paths of initially i-equivalent executions do not diverge, it must be extended to handle declassification. In the worst theoretical case such a reduction from security to safety would require constructing a full product of the initial program, including not only duplication of program data, but also including the cross product of control paths. This poses a significant practical problem for program

```
entry: eq i:V_i α(i):V_i a:x
       assume a:x
       goto b0
b0:    cp r:u a:x
       cp α(r):u a:y
       eq a:x a:y a:z
       assert a:z
       load[a0] r:u r:x
       load[a0] α(r):u α(r):x
       cp r:x a:x
       cp α(r):x a:y
       eq a:x a:y a:z
       assert a:z
       branch r:x b1
       branch α(r):x b1
       zero a:x
       zero a:y
       eq a:x a:y a:z
       assert a:z
       goto b2
       goto b2
b1:    ...
       goto b3
b2:    ...
       goto b3
b3:    cp r:r a:x
       cp α(r):r a:y
       eq a:x a:y a:z
       assert a:z
       store[m] r:r 0
       store[m] α(r):r 0
       zero a:x
       zero a:y
       eq a:x a:y a:z
       assert a:z
       goto exit
       goto exit
exit:  eq o:V_o α(o):V_o a:x
       assert a:x
       halt
```

Figure 2. The shadow product of the program in Figure 1.

verification tools, which typically possess adequate data abstractions to handle our duplication of program data, yet lack efficient means for representing the product of control paths. While the size of our shadow program is asymptotically linear in the size of the original program, the full product would be of polynomial size. This asymptotic increase is infamously known, in the general case of computing the product of transition systems, as the "state explosion problem" [14].

However, in practical cases (see Section 5) the divergence of control paths is limited to small statically-identifiable lexical scopes. For instance, consider the following program fragment

```
x = E;
if (x) { S1; y = 0; } else { S2; y = 1; }
z = y;
```

in which the value of `x` is private information, while `z` is declassified output, and `E`, `S1`, and `S2` are arbitrary expressions and statements. Such divergence would be permitted in finally d-equivalent executions since each control path yields distinct declassified output. When such situations are statically identifiable, we could contain the asymptotic increase in program size by computing the control product only for those limited lexical scopes, taking care—in case `S1` or `S2` themselves contain instructions that may leak—to use the data-only product construction from Section 3.1 whenever the control is in fact synchronized. For instance, denoting the shadow-copy of variables, expressions, and statements with primed variables, etc., the product of the above code fragment, not including its assertions, would be

```
x  = E ;
x' = E';
if (x == x') {
  if (x ) { P(S1; y = 0;) } else { P(S2; y = 1;) }
} else {
  if (x ) { S1 ; y  = 0; } else { S2 ; y  = 1; };
  if (x') { S1'; y' = 0; } else { S2'; y' = 1; };
}
z  = y ;
z' = y';
```

where we use `P(S)` to denote the construction $P_\times$ applied to the blocks encoding statement S. In addition, note that, in the presence of declassification, assertions must be delayed, since their validity may be conditioned by an equality on the final state of declassified variables. When using this extended product construction, we therefore collect assertions during the simulation. Assertions collected inside a declassified region (here the `if (x == x')` conditional statement) are conditioned by the declassified branching condition (here `x == x'`). For example, if the products `P(S1; y = 0;)` and `P(S2; y = 1;)` contain assertion `A`, then the simulation of the extended product collects assertion `(x == x') => A`.

4. Implementation of a Security Verifier

Using the reduction of Section 3 we have implemented a prototype tool which is capable of nearly-automatic verification of the actual C-code implementations of several constant-time algorithms. Before discussing the verification of these codes in Section 5, here we describe our implementation and outline key issues. Our implementation and case studies are all publicly available[3] and cross platform.

Our implementation is essentially a series of translations from the C source code to SMT[4] solver queries. Initially, the C source code must be annotated in order to distinguish public inputs, public outputs, and declassified outputs, requiring a modest manual effort. We then leverage the SMACK verification tool [29] to compile the annotated C source via Clang[5] and LLVM[6] to Boogie[7] code. We perform our reduction at the Boogie-code level, and apply the Boogie verifier to the shadow product of the original program, which performs verification using an underlying SMT solver. In some cases we require a minor amount of further manual effort in identifying simple loop invariants.

4.1 Security Annotations

Applying our prototype requires a modest amount of manual effort in annotating the public input and output values to entry-point procedures. We provide a simple annotation interface via the following C function declarations:

```
void public_in(smack_value_t);
void public_out(smack_value_t);
void declassified_out(smack_value_t);
```

where `smack_value_t` values are handles to program values obtained according to the following interface

```
smack_value_t __SMACK_value();
smack_value_t __SMACK_values(void* ary, unsigned count);
smack_value_t __SMACK_return_value(void);
```

where `__SMACK_value(x)` returns a handle to the value stored in program variable `x`, `__SMACK_values(ary,n)` returns a handle to an `n`-length array `ary`, and `__SMACK_return_value()` provides a handle to the procedure's return value. While our current interface does not provide handles to entire structures, non-recursive structures can still be annotated by annotating the handles to each of their (nested) fields. Figure 3 demonstrates the annotation of a decryption function from the Tiny Encryption Algorithm (TEA). The first argument `v` is a pointer to a message of two 32-bit words, while the second argument `k` is a pointer to a secret key.

4.2 Reasoning about Memory Separation

In many cases verifying security relies on reasoning about the separation of memory objects. For instance, if the first of two adjacent objects in memory is annotated as public input, while

[3] LINK TO REPOSITORY REMOVED FOR DOUBLE BLIND REVIEW

[4] Satisfiability Modulo Theories Library, `http://smtlib.cs.uiowa.edu`

[5] A C language family frontend for LLVM, `http://clang.llvm.org`.

[6] The LLVM Compiler Infrastructure, `http://llvm.org`.

[7] Boogie, `http://github.com/boogie-org/boogie`.

```
void decrypt_cpa_wrapper(uint32_t* v, uint32_t* k) {
  public_in(__SMACK_value(v));
  public_in(__SMACK_value(k));
  public_in(__SMACK_values(v, 2));
  decrypt(v, k);
}
```

Figure 3. The security annotations for the decrypt function of the Tiny Encryption Algorithm (TEA).

the second is not, then a program whose branch conditions rely on memory accesses from the first object is only secure if we know that those access stay within the bounds of the first object. Otherwise, if those accesses might within the second object, then the program is insecure since the branch conditions can rely on private information.

Luckily SMACK has builtin support for reasoning about the separation of memory objects. Internally, SMACK leverages an LLVM-level data-structure analysis [20] (DSA) in order to efficiently encode memory operations in Boogie. Concretely SMACK uses DSA to partition memory objects into disjoint regions so that the resulting Boogie program encodes memory as several disjoint map-type global variables rather than a single monolithic map-type global variable, which facilitates scalable verification of the resulting Boogie code. In many cases this separation of objects into regions already provides sufficient separation for verifying security.

However, since DSA is designed to be an extremely scalable analysis, it sacrifices precision in many situations. This lack of precision manifests in the unification of disjoint memory objects, or the complete collapse of useful information, for certain syntactic patterns on which DSA refuses to reason about precisely. For instance, DSA unifies otherwise-disjoint objects passed as arguments to function calls with the same call paths. DSA also collapses objects when separate offsets within are passed as arguments to the same function call.

In cases when DSA lacks the precision to separate public objects from private ones, we have taken two separate approaches to regain reasoning with separation. In many cases it is possible to modify the C source code to avoid syntactic patterns which are difficult for DSA to reason about. In cases when this is not desirable or possible, we annotate the C source code with additional assumptions on object separation using SMACK's `__VERIFIER_assume(E)` function to limit verification to executions for which the expression `E` evaluates to true. While manual effort is involved in either case, we believe that this particular effort is an artificial limitation of using DSA, and not one fundamental to our approach.

4.3 The Shadow Product of Boogie Code

The Boogie intermediate verification language (IVL) is a simple imperative language with well-defined, clean, and mathematically-focused semantics which is a convenient representation for performing our reduction. Conceptually there is little difference between performing our shadow product reduction at the Boogie level as opposed to the LLVM or machine-code level since the Boogie code produced by SMACK is isomorphic to the LLVM code, which is itself similar to machine code. Indeed our machine model of Section 2 is representative. Practically however, Boogie's minimal syntax greatly facilitates our code-to-code translation. In particular, shadowing the machine state amounts to making duplicate copies of program variables. Since memory accesses are represented by accesses to map-type global variables, accessing a shadowed address space amounts to accessing the duplicate of a given map-type variable.

Our prototype models observations as in Example 1 of Section 2, exposing the addresses used in memory accesses and the values used as branch conditions as observations. According to our construction of Section 3, we thus prefix each memory access by an assertion that the address and its shadow are equal, and prefix each branch by an assertion that the condition and its shadow are equal. Finally, for procedures with annotations, our prototype inserts assume statements on the equality of public inputs with their shadows at entry blocks, and assert statements on the equality of public outputs with their shadows at exit blocks.

4.4 Scalability of the Boogie Verifier

Since secure implementations, and cryptographic primitives in particular, do not typically call recursive procedures, we instruct Boogie to inline all procedures during verification. This avoids the manual effort that would otherwise be involved in writing procedure contracts. Due to the relatively-small size of such implementations, this has not led significant scalability issues, though in a few extreme cases discussed in Section 5, where scalability was an issue, we did manually write contracts for a select few impactful procedures. Additionally, in many of the implementations we verified we were able to unroll loops completely in order to avoid the manual effort in writing loop invariants.

For the implementations for which complete loop unrolling did not scale, we were able to compute most of the required loop invariants automatically. Intuitively this is possible because most of the required loop invariants were equalities between program variables and their shadowed copies. We compute the relevant set of variables by taking the intersection of variables live at loop heads with those on which assertions, e.g., inserted by our reduction, depend.

4.5 Handling Declassification

As discussed in Section 5 we encountered only few implementations which use declassified outputs. Accordingly, rather than automating the extended reduction outlined in Section 3.2 which handles declassification, in the few cases where the extended reduction was necessary, we performed it manually at the Boogie level with little effort.

5. Empirical Results

We evaluate our prototype on various examples,[8] mainly taken from cryptographic libraries. In most cases, the simple product construction without declassification is sufficient. We first discuss some fully automated results, then illustrate declassification and a stronger leakage model on a particular example and argue that our chosen backend is sufficiently robust to deal with these extensions to the product construction.

All execution times reported in this section were obtained on a 2.7GHz Intel i7 with 16GB of RAM. The tool currently assumes that the leakage trace produced by standard library functions `memcpy` and `memset` depends only on their arguments (that is, the address and length of the objects they work on, rather than their contents). Additional example-specific assumptions are clarified where relevant. Throughout this section, size statistics measure the size in LoC of the analyzed Boogie code (similar in size to the analyzed LLVM bitcode) after pruning but before inlining. When presenting time measurements, we separate the time taken to produce the product program (annotating it with the $\times$ symbol) from that taken to verify it: in particular, given a library, the product program can be constructed once and for all before verifying each of its entry points.

5.1 Fixed-Point Arithmetic

Our first example is the `libfixedtimefixedpoint` library, developed by Andrysco et al. [5] to mitigate several attacks due

[8] LINK TO REPOSITORY REMOVED FOR DOUBLE BLIND REVIEW

Function	Size	Time (s)
fix_eq	100	1.45
fix_cmp	500	1.44
fix_mul	2300	1.50
fix_div	1000	1.53
fix_ln	11500	2.66
fix_convert_from_int64	110	1.43
fix_sin	800	1.64
fix_exp	2200	1.62
fix_sqrt	1400	1.55
fix_pow	18000	(⊥) 9.88
fix_pow_fixed	18000	5.28

Table 1. Verification of libfixedtimefixedpoint. Sizes are in lines of LLVM.

to data-dependent leakage in the timing of floating point operations. The library itself is pretty small in size (ca. 4KLoC of C or 40,000LoC in LLVM) and implements a large number of fixed-point arithmetic operations, from comparisons and equality tests to exponentials and trigonometric functions. Core primitives are automatically generated parametrically in the size of the integer part: we verify code generated with the default parameters. As far as we are aware, this is the first application of formal verification to the libfixedtimefixedpoint library.

Table 1 shows verification statistics for part of the library. We verify all functions without any inputs marked as public,[9] but display only some interesting data points here. Our tool identifies a known issue in the fix_pow function. A corrected version of the function, fix_pow_fixed is then successfully analyzed. We discuss these two cases in more detail.

At some point during the execution of fix_pow, the code shown in Figure 4 is executed on a frac array whose contents are not public. Our tool successfully identifies that the branch on the value of frac[i] and the subsequent break leak information about the contents of frac. We modify the snippet as shown in Figure 5 to produce the fix_pow_fixed implementation. The fix simply uses the mask variable to keep track of whether the current loop iteration is meaningful or not and using its value, at each loop iteration, to multiplicatively mask updates to the state variables.

5.2 Simple cryptographic primitives

For our second set of examples, we consider small to medium sized implementations of cryptographic primitives: a standard implementation of TEA [37] (tea), an implementation of sampling in a discrete Gaussian distribution by Bos et al. [10] (rlwe_sample), parts of the libsodium library [9] (nacl_xxxx) and the curve25519-donna implementation of some elliptic curve arithmetic functions by Langley.[10] Although the curve25519-donna implementation was checked using Langley's ctgrind dynamic analysis tool,[11] this is the first static analysis result on a compiled version of the library.

The verification results for rlwe_sample does not cover its core random byte generator, essentially proving that this core primitive is the only possible source of leakage. In particular, if its leakage and output bytes are independent, then the implementation of the sampling operation is constant-time.

We now discuss the cause of the verification failure for nacl_sha256. We first recall that the SHA256 hash of a message is constructed by iteratively merging its constituent blocks into a state (initialized with a publicly specified initialization vector) via a *compression function*. Our tool successfully and quickly verifies the SHA256 compression function as implemented in the library (nacl_sha256), but fails to verify its iterated version, despite the simplicity of the construction. Observing the Boogie code reveals that the structure containing the state is encoded into a single *memory region*, rather than having its sub-objects (the length, current state, and remaining data) being stored in separate regions. Further observing the debugging output of the SMACK frontend shows that its DSA analysis collapses the structure into a single region for several reasons. First, the memset function is used to initialize it and zero it out. Replacing this single memset with separate calls for each field of the structure is sufficient to avoid this collapse. Further, the nacl_sha256 function calls nacl_sha256_block with arguments that point to two different offsets into the state. Modifying the code to avoid this particular issue would require introducing a copy operation, which is not necessarily acceptable in all contexts. However, it is possible to inform BOOGIE that the two memory objects remain separate despite the collapse using an assume statement as outlined in Section 4.2. In theory, such assume statements would be present (as verified assertions) as part of a proof of memory safety and could therefore be reused if they exist. We leave further investigation into the interaction of memory safety and leakage security as future work.

In addition, we note that these two particular causes of region collapse in DSA are artefacts of its optimizations. It may be interest-

```
uint64_t result = 0;
uint64_t extra  = 0;

for(int i = 0; i < 20; i++) {
  uint8_t digit = (frac[i] - (uint8_t) '0');
  if(frac[i] == '\0') {
    break;
  }
  result += ((uint64_t) digit) * pow10_LUT[i];
  extra  += ((uint64_t) digit) * pow10_LUT_extra[i];
}
```

Figure 4. Original code.

```
uint64_t result = 0;
uint64_t extra  = 0;
uint8_t  mask   = 0x01;

for(int i = 0; i < 20; i++) {
  uint8_t digit = (frac[i] - (uint8_t) '0');
  mask   &= (frac[i] != '\0');
  result += mask * ((uint64_t) digit) * pow10_LUT[i];
  extra  += mask * ((uint64_t) digit) * pow10_LUT_extra[i];
}
```

Figure 5. A constant-time variant.

Example	Size	Time (×)	Time
tea	200	2.33	0.47
rlwe_sample	400	5.78	0.65
nacl_salsa20	700	5.60	1.11
nacl_chacha20	10000	8.30	1.92
nacl_sha256_block	20000	27.7	4.17
nacl_sha256	20000	31.40	(⊥) 41.78
curve25519-donna	10000	10.18	456.97

Table 2. Verification of small- and medium-sized crypto examples. Sizes are in lines of LLVM, verification times are in seconds.

[9] We do not verify printing functions or conversions to and from floating point.

[10] https://code.google.com/p/curve25519-donna/

[11] https://github.com/agl/ctgrind

Example	Time (×)	Time
`mee-cbc-openssl`	10.6	18.73
`mee-cbc-nacl`	24.64	92.56

Table 3. Verification of MEE-CBC. Times in seconds.

ing to consider shifting the precision/performance tradeoff of DSA in the direction of precision when used for formal verification of small to medium programs rather than the "efficient compilation of millions of lines of code".

5.3 MEE-CBC

To further illustrate the scalability to large code bases, we now consider problems related to the MAC-then-Encode-then-CBC-Encrypt (MEE-CBC) construction used in the TLS record layer to obtain an authenticated encryption scheme. It is well-understood from the perspective of provable security [18, 27], but implementations of this construction have been the source of several practical attacks on TLS stacks via timing side-channels [1, 12].

We apply our prototype to two C implementations of the MEE-CBC decryption procedure, treating only the input ciphertext as public information. Table 3 shows the corresponding verification results. We extract the first implementation from the OpenSSL sources (version 0.9.8zg). It includes all the countermeasures against timing attacks currently implemented in the MEE-CBC component in OpenSSL as documented in [19]. We extract by hand the parts of the code that are of particular interest for MEE-CBC decryption (1000LoC of C, or 10000LoC in LLVM): i. decryption of the encrypted message using AES128 in CBC mode; ii. removing the padding and checking its well-formedness; iii. computing the HMAC of the unpadded message, even for bad padding, and using the same number of calls to the underlying hash compression function (in this case SHA-1); and iv. comparing the transmitted MAC to the computed MAC in constant-time. Our verification does not include the leaf functions (for the SHA1 compression function and AES-128), since they are not implemented in C and therefore proves that the only possible leakage comes from these leaf functions. (In practice, SHA1 is constant-time and AES-128 is constant-branching but makes secret-dependent memory accesses.)

As our second example, we consider a full hand-crafted 800LoC (in C, 20000LoC in LLVM) implementation of MEE-CBC that follows strict constant-time coding policies and includes the implementation of the primitives (taken from the NaCl library).

Our prototype is able to verify the constant-time property of both implementations–with only the initial ciphertext and memory layout marked as public–fully automatically. Perhaps surprisingly, our simple heuristic for loop invariants is sufficient to handle complex control-flow structures such as the one shown in Figure 6, taken from OpenSSL. It should be noted that neither of these examples could be handled by previous tools working on compiled code. In particular, the certified type system by Barthe et al. [7] is unable to handle local array variables and structures with mixed security levels.

5.4 Extended product

We now illustrate the flexibility of the product construction approach by considering a slightly modified version of the `mee-cbc-nacl` example. Instead of using a constant-time select and zeroing loop to return its result (as shown in Figure 7, where the return code `res` is secret-dependent and marked as declassified and `in_len` is a public input), we branch on the return code as shown in Figure 8. (The rest of the code is unmodified, and therefore constant-time.) In practice, however, that return code is made public when it is returned to the caller and branching on its value therefore reveals nothing that is not already public.

```
k = 0;
if (/* low cond */) { k = /* low exp */ }
if (k > 0)
{ for (i = 1; i < k / /* low var */; i++)
  { /* i-dependent memory access */ }
}
for (i = /* low var */; i <= /* low var */; i++)
{ /* i-dependent memory access */
  for (j = 0; j < /* low var */; j++)
  { if (k < /* low var */)
      /* k-dependent memory access */
    else if (k < /* low exp */)
      /* k-dependent memory access */
    k++;
  }
  for (j = 0; j < /* low var */; j++)
  { /* j-dependent memory access */ }
}
```

Figure 6. A complex control-flow structure from OpenSSL.

```
good = ~((res == RC_SUCCESS) - 1);
for(i = 0;i < in_len;i++) {
  out[i] &= good;
}
*out_len &= good;
```

Figure 7. Constant-time conclusion of MEE-CBC decryption.

```
if (res != RC_SUCCESS) {
  for(i = 0;i < in_len;i++) {
    out[i] = 0;
  }
  *out_len = 0;
}
```

Figure 8. Regular conclusion of MEE-CBC decryption.

To verify the variant that ends as described in Figure 8, we perform the extended product construction described in Section 3.2 declassifying only the displayed conditional. To do so, we first use a variant of our tool that delays assertions (simply by keeping track of their conjunction in a special variable) but otherwise produces the standard product program. We then replace the blocks corresponding to the potentially desynchronized conditional with blocks corresponding to the extended product construction that mixes control and data product. Finally, we insert code that saves the current assertions before the declassified region and restores them afterwards, making sure to also take into account the assertions collected inside the declassified region. The extended product program thus constructed verifies in slightly less than 2 minutes. The additional computation cost of verifying this non-constant-time version of the program may be acceptable when compared to the performance gains in the program itself—however minor: verification costs are one off, whereas performance issues in the cryptographic library are paid per execution.

6. Related Work

Our work follows a long development of countermeasures against timing attacks, and the verification of implementations of countermeasures.

6.1 Countermeasures against Timing Attacks

High-visibility attacks like Lucky 13 [1, 11] have shown that timing attacks can be explored in practice to break the security of pervasively used protocols such as TLS, and have lead practicioners to pay renewed attention to software countermeasures against timing attacks. Two prominent examples of this are the reimplementation of MEE-CBC decryption in openSSL [19], which enforces a constant-time coding policy as mitigation for the Lucky 13 attack, and the *defense in depth* mitigation strategy adopted by Amazon Web Services - Labs (AWS-Labs) in S2N [30], where various fuzzing- and balancing-based timing countermeasures are combined to reduce the amount of information leaked through timing.

Software countermeasures against timing attacks have a history that is almost as long as the discovery of the attacks themselves. Practitioners soon realized that, ideally, the control flow of cryptographic code should be independent of secret values [13]. For a long time, however, this strategy was perceived as implying an unnacceptable performance overhead, and less aggressive protection measures were adopted. Roughly, these consist in balancing the execution times in various control-flow paths, so that harmful timing variability is reduced in the overal execution time of the implementation [6]. Nevertheless, early work in the formalization and analysis of such countermeasures [22] clearly highlighted the difference between the two approaches: while a strict independence of control-flow from secret data ensures that the full sequence of executed instructions can be leaked to the adversary (the so-called program-counter model), the balancing approach provides protection against a much weaker (and harder to characterize) class of adversaries that can take coarse measurements of the execution time.

6.2 Verification of Timing-Attack Countermeasures

Several works have looked at the verification of countermeasures agains timing attacks, formalized as non-interference properties between secret inputs and leakage outputs instrumented in the target language semantics.

Information flow type systems have been used to enforce non-interference in different contexts [23, 24, 33, 34, 36]. Volpano and Smith [35] explored the use of type systems to protect programs against covert termination and timing channels. Constant-time coding policies have been more recently addressed in [7] using a type system implemented in Coq as an extension to the CompCert certified compiler. The compiler takes as input a C program whose inputs are annotated with security levels (public or secret). The program is compiled using CompCert, and the security levels are transferred to assembly level using a simple procedure written in Caml—technically, the analysis is done at the Mach level. The assembly program is then analyzed using two dataflow analyses. Because it is certified, successful runs of the analyses output a formal proof that the analyzed program is constant-time. Obtaining this proof does not put any additional burden on the user—except for marking program inputs as secret or public. However, the code must satisfy a number of restrictions in order to be analyzed using the dataflow analysis from [7].

The main challenge in designing these information-flow type systems is that they are often too conservative in practice – secure programs may be rejected. The tool presented in this work is not certified, but it has significant advantages with respect to the mechanism in [7]: i. it deals with off-the-shelf code without modification; ii. it is conservative in the sense that *public* data must be tagged, rather that secret data; and iii. its semantic-based approach implies that it will accept a much wider class of constant-time programs.

Leino and Joshi [21] were the first to propose a semantic approach to secure information flow, with several desirable features: i. a precise characterisation of security; ii. it applies to all programming constructs with well-defined semantics; and iii. it can be used to reason about indirect information leakage through variations in program behaviour (e.g. termination). Dufay et al. [15] have proposed an extension to JML to enforce non-interference through self-composition, allowing for a simple definition of non-interference for Java programs. However, the generated proof obligations are forbiddingly complex. Terauchi and Aiken [32] identified problems in the self-composition approach, arguing that automatic tools (like software model checkers) are not powerful enough to verify this property over programs of realistic size. The authors propose a program transformation technique for an extended version of self-composition. Rather than replicating the original code, the renamed version is interleaved and partially merged with it. Naumann [25] extended Terauchi and Aiken's work to encompass heap objects, presented a systematic method to validate the transformations of [32], and reported on the experience of using these techniques with the Spec# and ESC/JAVA2 tools.

The product construction we use in this work can be seen as an extreme specialization of interleaved self-composition that focuses specifically on constant-time verification, with surprising automation benefits. In particular, the effectiveness and simplicity of the loop invariant inference technique that we have implemented in our tool stands in contrast with the natural invariants [2–4] used in previous works relying on self-composition. Svenningsson and Sands [31] have also addressed control-flow non-interference using self-composition. These authors also considered the issue of declassification, enabling the formal verification that only controlled amounts of leakage can occur (e.g. the leakage of the hamming weight of a secret during a modular exponentiation). Our view of declassification is more limited, and is restricted to allowing security policy violations that can be traced down to output values revealed to an adversary. Our motivation for covering such use-cases is that it significantly enlarges the set of real-world programs that can be accepted by the analysis, whilst introducing a negligible overhead in the analysis.

References

[1] N. J. AlFardan and K. G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy, SP 2013*, pages 526–540. IEEE Computer Society, 2013.

[2] J. B. Almeida, M. Barbosa, J. S. Pinto, and B. Vieira. Verifying cryptographic software correctness with respect to reference implementations. In *FMICS'09*, volume 5825 of *LNCS*, pages 37–52, 2009. ISBN 978-3-642-04569-1.

[3] J. B. Almeida, M. Barbosa, J. S. Pinto, and B. Vieira. Deductive verification of cryptographic software. *ISSE*, 6(3):203–218, 2010.

[4] J. B. Almeida, M. Barbosa, J. S. Pinto, and B. Vieira. Formal verification of side-channel countermeasures using self-composition. *Sci. Comput. Program.*, 78(7):796–812, 2013. doi: 10.1016/j.scico.2011.10.008. URL `http://dx.doi.org/10.1016/j.scico.2011.10.008`.

[5] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 623–639. IEEE Computer Society, 2015. doi: 10.1109/SP.2015.44. URL `http://dx.doi.org/10.1109/SP.2015.44`.

[6] M. Barbosa and D. Page. On the automatic construction of indistinguishable operations. Cryptology ePrint Archive, Report 2005/174, 2005. `http://eprint.iacr.org/2005/174`.

[7] G. Barthe, G. Betarte, J. D. Campo, C. D. Luna, and D. Pichardie. System-level non-interference for constant-time cryptography. In G.-J. Ahn, M. Yung, and N. Li, editors, *ACM CCS 14*, pages 1267–1279. ACM Press, Nov. 2014.

[8] D. J. Bernstein. Cache-timing attacks on aes, 2005. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[9] D. J. Bernstein. Cryptography in NaCl, 2011. http://nacl.cr.yp.to.

[10] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 553–570. IEEE Computer Society, 2015. doi: 10.1109/SP.2015.40. URL http://dx.doi.org/10.1109/SP.2015.40.

[11] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005. doi: 10.1016/j.comnet.2005.01.010. URL http://dx.doi.org/10.1016/j.comnet.2005.01.010.

[12] B. Canvel, A. P. Hiltgen, S. Vaudenay, and M. Vuagnoux. Password interception in a SSL/TLS channel. In D. Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 583–599. Springer, Heidelberg, Aug. 2003.

[13] B. Chevallier-mames, M. Ciet, and M. Joye. Low-cost solutions for preventing simple side-channel analysis: Side-channel atomicity. *IEEE Transactions on Computers*, 53:760–768, 2004.

[14] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2001. ISBN 978-0-262-03270-4. URL http://books.google.de/books?id=Nmc4wEaLXFEC.

[15] G. Dufay, A. Felty, and S. Matwin. Privacy-sensitive information flow with JML. In *Automated Deduction - CADE-20*, pages 116–130. Springer, August 2005.

[16] E. Käsper and P. Schwabe. Faster and timing-attack resistant AES-GCM. In C. Clavier and K. Gaj, editors, *CHES 2009*, volume 5747 of *LNCS*, pages 1–17. Springer, Heidelberg, Sept. 2009.

[17] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, Heidelberg, Aug. 1996.

[18] H. Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In J. Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 310–331. Springer, Heidelberg, Aug. 2001.

[19] A. Langley. Lucky thirteen attack on TLS CBC. Imperial Violet, Feb. 2013. https://www.imperialviolet.org/2013/02/04/luckythirteen.html, Accessed October 25th, 2015.

[20] C. Lattner, A. Lenharth, and V. S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In J. Ferrante and K. S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 278–289. ACM, 2007. doi: 10.1145/1250734.1250766. URL http://doi.acm.org/10.1145/1250734.1250766.

[21] K. R. M. Leino and R. Joshi. A semantic approach to secure information flow. *LNCS*, 1422:254–271, 1998.

[22] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In D. Won and S. Kim, editors, *ICISC 05*, volume 3935 of *LNCS*, pages 156–168. Springer, Heidelberg, Dec. 2006.

[23] A. C. Myers. Jflow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.

[24] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.

[25] D. A. Naumann. From coupling relations to mated invariants for checking information flow. In *ESORICS'06*, volume 4189 of LNCS, pages 279– 296, 2006.

[26] D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. Cryptology ePrint Archive, Report 2002/169, 2002. http://eprint.iacr.org/2002/169.

[27] K. G. Paterson, T. Ristenpart, and T. Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In D. H. Lee and X. Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 372–389. Springer, Heidelberg, Dec. 2011.

[28] C. Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[29] Z. Rakamaric and M. Emmi. SMACK: decoupling source language details from verifier implementations. In A. Biere and R. Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 106–113. Springer, 2014. doi: 10.1007/978-3-319-08867-9_7. URL http://dx.doi.org/10.1007/978-3-319-08867-9_7.

[30] S. Schmidt. Introducing s2n, a new open source tls implementation, June 2015. https://blogs.aws.amazon.com/security/post/TxCKZM94ST1S6Y/Introducing-s2n-a-New-Open-Source-TLS-Implementation, Accessed October 25th, 2015.

[31] J. Svenningsson and D. Sands. Specification and verification of side channel declassification. In *FAST'09*, volume 5983 of *LNCS*, pages 111–125. Springer, 2009.

[32] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS'2005*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005.

[33] S. Tse and S. Zdancewic. A design for a security-typed language with certificate-based declassification. In *ESOP'05*, volume 3444 of *LNCS*, pages 279–294. Springer, 2005.

[34] J. A. Vaughan and S. Zdancewic. A cryptographic decentralized label model. In *IEEE Symposium on Security and Privacy*, pages 192–206. IEEE, 2007.

[35] D. M. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *CSFW*, pages 156–169. IEEE Computer Society, 1997.

[36] D. M. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT'97*, volume 1214 of *LNCS*, pages 607–621. Springer, 1997.

[37] D. J. Wheeler and R. M. Needham. TEA, a tiny encryption algorithm. In B. Preneel, editor, *FSE'94*, volume 1008 of *LNCS*, pages 363–366. Springer, Heidelberg, Dec. 1995.