Detecting Timing Side-Channels in Executables

Bachelor Thesis

Jannik Friemann

June 2020

DOI: https://doi.org/10.15480/882.2852

Supervisors:
Prof. Dr. Dieter Gollmann
M.Sc. Marc Gourjon

Hamburg University of Technology
Security in Distributed Applications
`https://www.tuhh.de/sva`
Am Schwarzenberg-Campus 3
21073 Hamburg
Germany

Declaration

I, Jannik Friemann, solemnly declare that I have written this bachelor thesis independently, and that I have not made use of any aid other than those acknowledged in this bachelor thesis. Neither this bachelor thesis, nor any other similar work, has been previously submitted to any examination board.

Hamburg, June 9, 2020

Jannik Friemann

Abstract

Constant-time implementations are a popular approach for defending against cache-timing attacks. It is necessary to verify the resulting executables of such implementations, because the compiler might introduce timing side-channels during code optimization, leaving the program vulnerable even though the source code has no indication of possible timing side-channels. This thesis proposes a novel approach for formally verifying executables to be constant-time, by developing a type system for scVerif's low-level, assembly-like intermediate language, which features explicit leak statements. The resulting type checker can detect timing side-channels for arbitrary leakage models, including data-dependent instruction timings. However, the implementation needs further work before it can be applied on real world cryptographic implementations. Once a better implementation of the approach exists, it can easily be used for automatic checks of executables for cache-timing side-channels. Users only have to specify the initial secrecy of every variable using easy to use source code comments.

Contents

1 Introduction

The runtime behavior of a given implementation might differ for some inputs. This is important since it can be exploited to learn confidential values by observing the runtime behavior on different (confidential) inputs.

An attacker can perform attacks on the program and potentially extract its entire secret key, if the timing differences can be correlated to confidential data. Such cache-timing attacks have already been applied on implementations of AES [11], RSA [38], DSS and Diffie-Hellman [21]. Even worse, these measurements can be done remotely [15, 13]. Information leaking through timing characteristics of programs is also called *timing side-channel*.

To prevent timing side-channels, hardware-based and software-based countermeasures are used. Software-based countermeasures are usually easier to deploy than hardware-based countermeasures and are therefore often preferred, with *constant-time programming* (Section 2) being the most popular defense [10].

There are many tools to verify the correctness of constant-time implementations. Some work by analyzing the source code of programs [6, 3], while others are implemented as compiler plugins [5, 30]. But they are not enough, because there is no formal guarantee of the software-based countermeasures not being optimized away by the compiler [31]. There are even examples, where constant-time implementations are optimized away (see Section 2.2). This means that the results of the software-based countermeasures may not apply to the final executable. Additionally, they leave out instruction-based timing side-channels (Section 2), that are related to the instruction set architecture (ISA) of the machine running the program, of which the tools have no knowledge about.

Some tools do analyze the final executable after compilation [2, 29] to resolve both issues. Unfortunately, they do not provide any formal guarantees like the previous ones[1]. The tool from [29] for example gives no guarantee, because it only tells the user whether all timing differences, for one given test vector and sample size, are within a certain threshold (t-test).

So there is a gap between formal high-level tools and informal low-level tools. Existing tools can either prove the correctness of constant-time implementations for high-level languages with no guarantee for the executable *or* work with executables without any formal guarantees.

This thesis intends to narrow the gap by considering how cache-timing side-channel resilience can be verified on executables for specific hardware architectures. Solving this problem allows to reduce the gap between high-level and low-level tools by providing the formal guarantees offered by the high-level tools with the precision of the low-level tools. A novel approach needs to capture the data-dependent

[1] if the compiler is to be trusted

timing of instructions as well.

Section 3 explains how a type system (TS), designed for verifying constant-time implementations written in a high-level language, can be applied to a low-level, assembly-like language. The TS is extended to capture side-channel leakages of variable-timing instructions. Its type checker cannot infer the initial security types of all variables itself, Section 4 completes the approach by providing different methods for aiding the type checker with this problem.

2 Constant-Time Programming

Constant-time programming means writing software in such a way that the timing side-channel behavior of the program does not depend on secret data. It guards against remote attacks as well as attacks from the same machine, in a potentially concurrent execution (multithreading on same processor core) [8], if implemented properly. There are three main causes for timing side-channels:

Data-dependent control-flow can leak secret data when the conditional expression of a control-flow statement depends on secret data. This can leak information in three ways: An attacker can derive which branch was taken, if one branch takes longer to compute than the other. This way the attacker is able to deduce some additional information about the secret data that was used in the conditional expression. Furthermore, there is *explicit information-flow* when the value of secret data is assigned to a publicly available variable, e.g. **if** $sec > foo$ **then** $pub = sec$. An attacker can then directly read the secret value through *pub*, if it is used in an operation that leaks its value through a timing side-channel or provided as an output to the attacker. There is also *implicit information-flow*, where the value of a public variable depends on secret data, e.g. **if** $sec > foo$ **then** $pub = 42$ **else** $pub = 17$. The attacker might not know the value of *sec* by observing *pub* through a side-channel, but can still derive whether it was bigger than foo or not. Furthermore, if the variable is not only observable but also modifiable, the attacker can initially set it to 0 and then increment it, until the condition is no longer fulfilled and the variable gets assigned to another value. At this point the attacker can conclude that *sec* and *pub* had the same value before the assignment.

Data-dependent memory accesses can leak secret data when it is used as an index while accessing an array or a collection, i.e. a memory access happens. Here, some values might be in a cache reducing the access time drastically or are not in a cache, resulting in the opposite. The timing difference between a cache-hit and cache-miss can vary by up to a factor of 100, for some hardware architectures [24]. This behavior has already been exploited for cache-timing attacks on AES [11], RSA [38], DSS and Diffie-Hellman [21].

Data-dependent instruction timings of the hardware emerge when instructions like multiplication or division behave differently depending on their operands. For division, for example, the time to compute an operation can almost double in extreme cases [19]. Even though the difference in cycles might be relatively small, if the program involves heavy use of these or similar operations, this will add up. In practice, these are often ignored, because they are entirely hardware-dependent and difficult to detect. What might be a constant-time operation on one ISA, might not be constant-time on another.

Furthermore, the manuals of ISAs can be incomplete in regards to timing characteristics. It is possible that an operation is constant-time for most operands, but for some special cases that are not mentioned in the manual, the instruction is not constant-time [16].

In this setting constant-time programming means that programs do not branch on secret data, do not perform memory accesses based on secret data, nor use instructions with data-dependent timings for computations on secret data.

2.1 Countermeasures against Timing Side-Channels

This section gives a brief overview of countermeasures against cache-timing attacks and how typical constant-time implementations look like.

Preventing secret-dependent control-flow Secret-dependent control-flow can be replaced by arithmetic expressions and masks.

Listing 2.1 shows a function that returns either the value of *a* or *b* depending on whether *coin* is 1 or 0 respectively. It is not constant-time, because it branches on the secret value *coin* causing an implicit information-flow to the return value of the function.

```
1 /**
2   * @param a public
3   * @param b public
4   * @param coin secret
5   * @return public
6   */
7 int select (int a, int b, int coin) {
8   return coin ? a : b;
9 }
```

Listing 2.1: Conditional assignment branching on secret data

If we rewrite it to Listing 2.2, there is no more branching at all and consequently no implicit information-flow from *coin* to the return value. However, it is still not guaranteed to be constant-time, because it uses multiplication.

```
1 /**
2   * @param a public
3   * @param b public
4   * @param coin secret
5   * @return public
6   */
7 int mul_select (int a, int b, int coin) {
8   return a*coin + b*(1-coin);
9 }
```

Listing 2.2: Conditional assignment without branching

Preventing secret-dependent instruction timings Instructions with varying timing characteristics should generally be avoided, if possible. For some instructions there are constant-time versions like “cmov” for “mov”, the x86 move instruction [14]. AES even got its own instruction set (AES-NI) on Intel CPUs, making cache-timing attacks on it near to impossible [26].

Coming back to our previous example, Listing 2.2: For multiplication the cycle count can depend on its operands [19] and both multiplications depend on secret data. We cannot make a formal argument about its runtime without knowing the target machine the code will run on. It might coincidentally be constant-time, because the instruction could be constant-time on that particular ISA, but it might as well not be. To be totally safe, we need to rewrite the function once more to a version like Listing 2.3.

```
1  /**
2   * @param a public
3   * @param b public
4   * @param coin secret
5   * @return public
6   */
7  int bit_select(int a, int b, int coin){
8    int mask = -coin;
9    return (a & mask) | (b & ~mask);
10 }
```

Listing 2.3: Constant-time implementation of Listing 2.1

Now, the function is constant-time, because it does not branch (on secret data), does not feature any “leaking” instructions with secret data, nor performs any memory accesses based on secret data. Thus, it is resilient against cache-timing attacks.

Preventing secret-dependent memory accesses Secret-dependent memory accesses can be avoided by iterating over the entire memory region (e.g. array or collection) and then performing the same rewriting we did in the conditional assignment example of Listing 2.3. Iterating over lots of elements is pretty inexpensive, however, all the arithmetic computations impose an overhead increasing the total accesses time noticeably. An alternative to this would be to use *stealth memory*. It can be used to grant processes designated cache lines that no other processes can access [28]. This way, attacks like *Flush+Reload* [37], *Evit+Time* and *Prime+Probe* [27], targeting processes on the same machine using shared caches, no longer work, while still having the performance benefit of using fast caches.

Consider the function given in Listing 2.4, it is merely a wrapper for the “[]”-operator used to access arrays. Furthermore, it is not constant-time, because it uses secret data for memory accesses.

```
1  /**
2   * @param array public 4
3   * @param index secret
4   * @return public
5   */
6  int arrayAccess(const int array[4], int index){
7   return array[index];
8  }
```

Listing 2.4: Wrapper function performing array accesses

Using the previous mentioned approach, we can rewrite it to Listing 2.5 by accessing the entire array and using a mask to write the value of the desired position into the return value.

```
1  /**
2   * @param array public 4
3   * @param index secret
4   * @return public
5   */
6  int ct_arrayAccess(const int array[4], int index){
7    int result = 0;
8    for(int i=0; i<4; i++){
9      int value = array[i];
10     // result = ct_select(value, result, i==index)
11     int isCorrectPosition = i == index;
12     int mask = 0 - isCorrectPosition;
13     result = (value & mask) | (result & ~mask);
14   }
15   return result;
16 }
```

Listing 2.5: Constant-time array access

2.2 Verifying Constant-Time Implementations

Just applying these rewritings is not enough and can be quite error-prone. When trying to fix a side-channel vulnerability in TLS [25], a new vulnerability called *Lucky 13* [4] was introduced to OpenSSL. Trying to fix Lucky 13, a third vulnerability [32] then got introduced [14]. Two years after the discovery of Lucky 13, Eisenbarth *et al.* discovered a new side-channel not considered by [4] and thus were able to "resurrect" Lucky 13 again [20]. This is a prime example showing that it is extremely important to verify constant-time implementation instead of manual security assessment. It remains questionable whether these security vulnerabilities could have been detected with fuzzing techniques.

There are multiple tools for verifying programs to be constant-time [5]. They all work on different stages of a programs life cycle. Some analyze the source code (e.g. C) statically [6, 3], others are implemented as compiler plugins [5, 30]. All of these share a common problem: they don't check the resulting program assuming that the compiler behaves nicely.

General-purpose compilers are aware of safety properties, e.g dereferencing a null-pointer and will warn the user about them. Information-flow cannot be reduced to safety properties in general [18], but constant-time is a carefully refined safety property, which is eligible to type checking approaches [5, 12]. However, general purpose compilers do not do this out of the box. They require special plugins like [14, 5] to do so. Thus, for most compilers, it may happen that software-based countermeasures get optimized away to code that is no longer constant-time. Consider the following example from [14]:

When compiling Listing 2.3 with the flags *-O2 -m32 -march=i386 -static-libgcc*, gcc (Listing 2.6) and clang (Listing 2.7) will produce different assemblies. While the assembly from gcc mirrors the source code implementation,

```
1  mov     0x8(%esp),%ecx
2  mov     0x4(%esp),%eax
3  xor     %ecx,%eax
4  movzbl  0xc(%esp),%edx
5  neg     %edx
6  and     %edx,%eax
7  xor     %ecx,%eax
8  ret
9  xchg    %ax,%ax
```

Listing 2.6: Listing 2.3 compiled using *gcc -O2 -m32 -march=i386 -static-libgcc*

clang optimized the entire computation (countermeasure) away and replaced it with a conditional jump.

```
1  cmpb    $0x0,0xc(%esp) ; conditional
2  jne     11ae           ; jump!
3  lea     0x8(%esp),%eax
4  mov     (%eax),%eax
5  ret
6  lea     0x4(%esp),%eax
7  mov     (%eax),%eax
8  ret
```

Listing 2.7: Listing 2.3 compiled using *clang -O2 -m32 -march=i386 -static-libgcc*

In this example, the compiler transformed the code back to the version the countermeasure tried to mitigate.

While Barthe *et al.* developed a general method for verifying compiler optimizations to preserve constant-time countermeasures [10], their method currently has not been applied to sophisticated, real-world compilers [31]. Thus, it is necessary to check the resulting executable (or rather its disassembly) after the compilation. Without, there is no formal guarantee that the result of a source code analysis applies to the final executable.

2.3 Constant-Time as a Formal Security Property

In order to verify constant-time implementations, we need to formally define what it means for a program to be constant-time. Before doing so, it is necessary to explain what leakage models are.

We will follow a similar definition to the one presented in [5]:

Definition 1 (Leakage Model)**.** *A leakage model consists of assumptions on what kind of instructions leak which information.*

In the context of *timing* side-channels, this means that leakage models describe which instructions have a varying influence on the runtime of a program and consequently leak information. The most frequently used ones model the following side-channel behaviors [10]:

1. The *program counter model (PCM)* models *data-dependent control-flow*, by assuming that all evaluated control-flow guards of a program are leaked.

2. The *memory obliviousness model (MOM)* models *data-dependent memory accesses*, by assuming that the address of every memory access is leaked.

3. The *constant-time policy (CTP)* is the result of combining PCM and MOM.

4. The *cost obliviousness model (COM)* models *data-dependent instruction timings* by assuming that the number of cycles each instruction takes during execution is leaked.

5. The *constant-time with cost model (CTC)* is the result of combining CTP and COM.

In the following constant-time always refers to Definition 2 inspired by [5]:

Definition 2 (Constant-time)**.** *Let P program, $\Im(P)$ the set of all possible secret inputs for P and $\lambda_m(P,x)$ all observable leakages of $P(x)$ within the leakage model m.*

$$P \text{ constant-time} \iff \forall x,y \in \Im(P) : \lambda_m(P,x) = \lambda_m(P,y).$$

A program is constant time if and only if the observable leakages for any two executions with different secret inputs are indistinguishable. It is also said that secret data is non-interfering with the leakages and thus with the runtime of the program, according to the specified leakage model.

This ensures that the behavior of the program, which is observable through timing side-channels, cannot be correlated to secret data. The program's runtime may still vary depending on its public inputs, but an attacker cannot exploit it, because the entire observable behavior is only correlated to the attacker's own manipulations.

This thesis uses *CTC* as the default leakage model, if nothing else is mentioned. Using this leakage model guarantees that a program is protected from cache-timing attacks, if it is constant-time [8].

Consider Listing 2.8 as an example.

```
1   /**
2    * @param a secret
3    * @param b public
4    * @return public
5    */
6   bool stringsAreEqual (string a, string b) {
7       for (int i = 0; i < a.length(); i++) {
8           if(a[i] != b[i]) return false;
9       }
10      return true;
11  }
```

Listing 2.8: Compares if two strings are equal

This program is not constant-time, because it branches on secret data (a). To prove this, we need to find two secret inputs causing different leakages[1]. For the first input, calling *stringsAreEqual("secret", "foobar")*, the program starts by comparing "f" and "s", notices that the characters differ, leaks $false$ according to the PCM and aborts the comparison early (line 8). The total leakages are $[false]$. Whereas for the second input, calling *stringsAreEqual("foopar", "foobar")*, the first three characters are equal, so three times $true$ gets leaked. The characters at the fourth position don't match, so $false$ is leaked and the comparison gets aborted. Here, the total leakages are $[true, true, true, false]$.

Using the notation from definition 2, we may simply write:

$$\tau_{CTC}(stringsAreEqual(\text{"}foobar\text{"},\text{"}secret\text{"})) = [false]$$

$$\neq$$

$$\tau_{CTC}(stringsAreEqual(\text{"}foobar\text{"},\text{"}foopar\text{"})) = [true, true, true, false]$$

We found two secret inputs, "secret" and "foopar", causing different leakages. It proves the program not being constant-time and consequently vulnerable to cache-timing attacks. This small example does not prove but demonstrate that non-constant-time programs are also non-constant-time within the used leakage models. It is therefore assumed that the leakage models reflect timing side-channel behavior, emerging in practice, sufficiently. If a program is constant-time within a model, then the executable is constant-time in as well, making model-based formal methods appropriate for verifying implementations.

[1] leaks from the loop checks are omitted for readability

3 Verifying Constant-Time Implementations

The previous section shows how constant-time can be formally expressed. It then demonstrates how this definition can be used to prove that a program is not constant-time, by providing two different secret inputs for which the resulting leakages differed. Proving programs to (not) be constant-time can be automated. In order to show that a program is constant-time, it is necessary to show that for all possible secret inputs, the resulting leakages for every leakage model stay the same, i.e. secret data is non-interfering with the resulting leakages statements. Therefore, all leakages can only be caused by public data. This is the same as saying that leaking statements can only be called with public inputs. By viewing secret and public as security types (taints), this problem can be considered as a type checking problem. Instead of checking if the types of a function's parameters match the signature (or parameter requirements), we check if the secrecy of all values matches the function's secrecy requirements. In doing so, we prove *secure information-flow*.

3.1 Constant-Time and Secure Information-Flow

The TS of [35] was the first type system for verifying secure information-flow. It can be used to verify non-interference between all secret and public data in a program. In other words, it gives us a formal guarantee that if a program can be typed with the rules of the type system (i.e. it is *typeable*), its secret variables don't influence its public variables and vice versa. It establishes a formal proof, where the application of typing rules are the steps of the proof.

Definition 3 (Type System)**.** *Let $T = \{secret, public\}$ with $public \leq secret$ be a set of ordered security types, where public is a lower type than secret*[1]*. Let $\tau : S \to T$ be a mapping from statements to security types. Then (T, τ) is a type system.*

Example how a type checker for the TS of [35] (TS0) would work. Let τ be a simplified version of the original typing rules consisting of the following rules:

$$\frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t'}{\Gamma \vdash op(e_1, e_2) : max\{t, t'\}}(arithmetic) \tag{3.1}$$

$$\frac{\Gamma \vdash e : t \qquad \Gamma \vdash e' : t' \qquad t' \leq t}{\Gamma \vdash e \leftarrow e' : t}(assignment) \tag{3.2}$$

$$\frac{\Gamma \vdash e : t \qquad \Gamma \vdash c : t' \qquad \Gamma \vdash c' : t'' \qquad t \leq t' \wedge t \leq t''}{\Gamma \vdash \textbf{if } e \textbf{ then } c \textbf{ else } c' : max\{t', t''\}}(if) \tag{3.3}$$

[1] T is a lattice

We will use the program "**if** $x = 1$ **then** $y \leftarrow 1$ **else** $y \leftarrow 0$" with $\tau(x) = secret$ as an example. The checker identifies that there is a statement of the form "**if** *expression* **then** *command* **else** *other_command*". According to Rule 3.3, we first have to type the expression "$x = 1$". $\tau(x) = secret$ and $\tau(1) = public$. Using Rule 3.1 the entire expression gets typed as $max\{secret, public\} = secret$. Now, Rule 3.3 constrains both *command* and *other_command* to have at least a type of *secret*, to avoid implicit information-flow. Otherwise an attacker could deduce the value of x, if y would be public, because the value of y depends on the value of x. Consequently, the type checker needs to verify that *command* ("$y \leftarrow 1$") and *other_command* ("$y \leftarrow 0$") can be typed as *secret*. For assignments this is only true when the left-hand side has type *secret* (Rule 3.2). Thus, if $\tau(y) = secret$, the program is typeable ensuring that secret data is non-interfering with public data, else the opposite is verified.

The property verified by the typing rules is not what we are looking for. The main purpose of the type system is to verify secure information flow. Like constant-time, this can be expressed as a form of non-interference. However, it does not mean that they are equal or that TS0 implies constant-time, because the PCM being a part of constant-time is more restrictive than secure information-flow, when it comes to branching. Secure information-flow allows branching on secrets, as long as both branches contain only statements that are typed as secret. The PCM disallows branching on secrets entirely, to avoid unbalanced branches.

Thus, a program can be information-flow secure in this TS but not be constant-time in practice: "**if** $input[0] = secret[0]$ **then** $value = input[0] + input[0] + secret[2]$ **else** $value = 7$" is typeable assuming *value* is *secret*. However, from a constant-time perspective, this program is leaking the value of $secret[0]$, because the **if** and the **else** branches are not balanced. The **if** branch will take significantly longer to compute, so an attacker can deduce the value of $secret[0]$ if he detects that the longer branch was taken based on the time the program takes to compute.

In theory, branching on secrets is acceptable, as long as both branches take the exact same number of cycles to compute and have the same memory access pattern. A static analysis cannot compute the amount of cycles without profound knowledge of the underlying hardware architecture, because the number of cycles may be data-dependent (Section 2). The simplest way to resolve this issue would be to ban secret-dependent branching entirely, because it is highly unlikely that both branches take the same amount of cycles and have the same observable side-channel, e.g. effect on registers, memory, caches and timing. This is what is often done in constant-time implementations.

So far, what we have stated only applies to the PCM. We have to consider MOM as well. Now, both branches also have to perform the same memory accesses keeping the caches in sync, to prevent cache-timing attacks. In order to check this, the type checker has to be able to handle memory accesses. All compiled programs get ported to assembly featuring memory access based on addresses (pointers to memory). A static type checker cannot deduce where a pointer points to. This impedes typing memory access which is absolute crucial, because they are a core feature of every programming language. Detecting cache-timing attacks is not possible without this knowledge.

In summary, TS0 is not suitable for detecting timing side-channels. Instead we would need a type system that is capable of also dealing with both memory accesses as well as hardware based leakages.

Following, we will investigate how such a type system could be designed and implemented.

3.2 Verifying Low-Level Code

Section 2.2 already motivated why its important to check the final executable over a programs source code. After looking at how type systems can be used for verifying secure information-flow in the previous section, we will now look at how such a type system for a high-level language can be ported to a low-level, assembly-like language. It will additionally be modified to also comply with CTC.

3.2.1 Introduction to scVerif

Instead of starting from scratch, we will base our work on scVerif [9], a tool for verifying side-channel resilience of executables for flexible leakage models. We can use it to analyze executables by providing it with the executable's disassembly. This of course assumes that the program generating the disassembly can be trusted. Currently, scVerif's only featured hardware model is the ARM Cortex-M0+ microcontroller. Therefore, we will use this as our target platform. scVerif uses its own domain specific language (DSL) called IL, as in "intermediate language".

It generates IL programs from assembly and then works entirely on IL, similar to how the LLVM compiler infrastructure works entirely on LLVM IR [22]. The special feature of IL is that it features explicit leak statements, which can be used to model the side-channel behavior of every single hardware instruction in great detail. Listing 3.1 shows how to instruct scVerif that every XOR[2] leaks its result (Line 3) and the first operand (Line 4).

```
1   macro eor2_leak (w32 op1, w32 op2)
2   {
3     leak eorResult(op1 ^w32 op2);
4     leak eorFirstOperand(op1);
5   }
```

Listing 3.1: Example macro demonstrating how IL uses explicit leak statements for modeling side-channel behavior

So far, scVerif was primarily used for reasoning about power and electromagnetic side-channels. This thesis extends it for timing side-channels, by adding a type checker for verifying that IL programs are constant-time.

An executable's IL version is assumed to be semantically identical, as no semantical modifications are performed during the conversion. Therefore, this thesis assumes that if the IL is (not) constant-time, the executable is (not) as well.

3.2.2 A Constant-Time Type System for IL

This section explains the initial typing rules for IL (TS1). It is a modified version of TS0, where some rules were changed to comply with CTC. A similar type system can be found in [8] for CompCert's

[2] the instruction name is "eor", for "exclusive or"

Mach IR [23]. TS1 still has some of the problems mentioned in Section 3.1. For a better understanding, we will first look at the pure type system and then go over the modifications needed to fix the aforementioned problems.

IL's syntax can be seen in figure 3.1.

$$
\begin{array}{rcl}
\chi & ::= & x \mid x[e] \mid \langle e \rangle \\
e & ::= & \chi \mid n \in \mathscr{Z} \mid l \mid o(e_1,\ldots,e_j) \\
i & ::= & \chi \leftarrow e \mid \texttt{leak}\,\{e_1,\ldots,e_j\} \mid m(e_1,\ldots,e_j) \\
 & & \mid \texttt{label} \mid \texttt{goto}\ e \\
 & & \mid \texttt{if}\ e\ \texttt{then}\ c_t\ \texttt{else}\ c_f \mid \texttt{while}\ e\ \texttt{do}\ c \\
c & ::= & i* \\
g & ::= & \textbf{var}\ x \mid \textbf{macro}\ m(x_1,\ldots,x_j)\ x_1,\ldots,x_j\ \ \{c\}
\end{array}
$$

Figure 3.1: Simplified syntax of the intermediate language scVerif uses, where n ranges on integers, x on variables, m on macro identifiers, o on operations and l on label identifiers [9].

We will use this syntax for designing the type system, by adding a typing rule for every derivation rule of it.

Rules for commands: Commands $c = (i_1,\ldots,i_n)$ are lists of instructions. The type of a command is the maximum of all instructions (Rule 3.4).

$$
\frac{\Gamma \vdash i_1 : t_1 \qquad \ldots \qquad \Gamma \vdash i_n : t_n}{\Gamma \vdash c : max\{t_1,\ldots,t_n\}}(command) \tag{3.4}
$$

Rules for instructions: For assignments, it is not allowed to write secret data into public variables to avoid explicit information flow (Rule 3.5).

$$
\frac{\Gamma \vdash \chi : t \qquad \Gamma \vdash e : t' \qquad t' \leq t}{\Gamma \vdash \chi \leftarrow e : t}(assignment) \tag{3.5}
$$

For leak statements leak $\{e_1,\ldots,e_n\}$, the expressions $e_1,\ldots,e_n$ that get leaked have to be public (Rule 3.6).

$$
\frac{\Gamma \vdash e_1 : public \qquad \ldots \qquad \Gamma \vdash e_n : public}{\Gamma \vdash leak\{e_1,\ldots,e_n\} : public}(leak) \tag{3.6}
$$

Else we would allow what we are trying to prevent. If one of the leaking expressions has type *secret*, it will result in a type error proving the program to be not constant-time.

Calling a macro requires no typing rule, because scVerif can inline macro calls and we therefore assume that all macros are inlined.

While secure information-flow allows branching on secret data, we will comply with the PCM and

ban it entirely (Rule 3.7 and 3.8).

$$\frac{\Gamma \vdash e : public \qquad \Gamma \vdash c_t : t_t \qquad \Gamma \vdash c_f : t_f}{\Gamma \vdash \textbf{if } e \textbf{ then } c_t \textbf{ else } c_f : max\{t_t, t_f\}}(if\ else) \tag{3.7}$$

$$\frac{\Gamma \vdash e : public \qquad \Gamma \vdash c : t}{\Gamma \vdash \textbf{while } e \textbf{ do } c : t}(while) \tag{3.8}$$

Thus, the expression e in both, **if** e **then** c_t **else** c_f (Rule 3.7) and **while** e **do** c (Rule 3.8), has to be public. This prevents implicit information flow.

Rules for expressions: Integers literals are known at compile time and don't change between different runs of the same program. Therefore, they cannot depend on secret data and are always public (Rule 3.9).

$$\frac{\Gamma \vdash n \in \mathscr{L}}{\Gamma \vdash n : public}(integer) \tag{3.9}$$

The same statement holds for static labels as well (Rule 3.10).

$$\frac{\Gamma \vdash l \text{ label}}{\Gamma \vdash l : public}(label) \tag{3.10}$$

Operators $op(\ldots)$ applied to expressions $e_1, \ldots, e_n$ compute a value with the maximum of its operators types as return type (Rule 3.11).

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \ldots \qquad \Gamma \vdash e_n : t_n}{\Gamma \vdash op(e_1, \ldots, e_n) : max\{t_1, \ldots, t_n\}}(operators) \tag{3.11}$$

This is necessary for tracking dependencies. Consider the following example: "$a \leftarrow public + secret$; **if** a **then** c **else** c'". a clearly depends on secret data and is later used as a guard for a branch. Without Rule 3.11, the type checker would not be able to recognize that the guard depends on secret data. The rule has one exception: $a \oplus a = 0$. When the operator is XOR and both operands are the same, the result will always be zero and therefore public.

Rules for state elements: The type of a variable x is of course the secrecy of its value (Rule 3.12).

$$\frac{\Gamma \vdash x \text{ variable with } \tau(x) = t}{\Gamma \vdash x : t}(variable) \tag{3.12}$$

The same applies to elements of arrays, however, it is not allowed that the expression e for accessing

an array x depends on a secret (Rule 3.13).

$$\frac{\Gamma \vdash e : public \qquad x \text{ array}}{\Gamma \vdash x[e] : \tau(x[e])}(array) \tag{3.13}$$

This would violate MOM. The same is also true for accessing a location in memory based on an expression e (Rule 3.14).

$$\frac{\Gamma \vdash e : public}{\Gamma \vdash < e >: \tau(< e >)}(memory\ access) \tag{3.14}$$

TS1 is able to detect secret-dependent control-flow, as in Listing 2.1, as well as instructions leaking secret data, as in Listing 2.2. However, both only work, if the code works entirely on the stack. As mentioned in Section 3.1, a pure type checker is incapable of handling pointers. Thus, modifying Listing 2.1 to Listing 3.2 causes the type checker to falsely label the code as constant-time and requires to take special care of memory accesses.

```
1 /**
2   * @param a public
3   * @param b public
4   * @param which secret 1
5   * @return public
6   */
7 int select (int a, int b, int *which) {
8   return *which ? a : b;
9 }
```

Listing 3.2: Conditional assignment branching on a pointer to secret data

Listing 3.3 is also labeled falsely for the same reason, even though it uses the leaking multiplication instruction with secret data.

```
1 /**
2   * @param a public
3   * @param b public
4   * @param which secret 1
5   * @return public
6   */
7 int mul_select (int a, int b, int *which) {
8   return a*(*which) + b*(1-(*which));
9 }
```

Listing 3.3: Conditional assignment without branching

3.3 Type Checking Execution Traces

The previous section shows how to adopt a type system for secure information-flow to a low-level, assembly-like language. However, the corresponding static type checker was unable to deal with

pointers. There are different approaches to aid it with this problem. For example, [8] suggests using a points-to analysis [33] in combination with the type checker to resolve the pointer problem. Albeit, scVerif is capable of partially evaluating the execution of IL code. In doing so, it rewrites every memory access to its target location. This means, every memory access is rewritten to accessing variables, so the checker only has to handle variables again. It is already able to do this. We can circumvent the pointer problem by first letting scVerif partially evaluate the IL program and afterwards type check the resulting execution trace. In doing so, the checker is now able to correctly label Listing 3.2 and Listing 3.3 as insecure.

However, memory accesses are not the only parts getting rewritten. The entire control-flow is evaluated as well. The final execution trace no longer contains control-flow. This makes detecting secret-dependent control-flow, as in Listing 2.1, impossible.

This approach also has another negative side-effect. The checker only has access to the last location a pointer points to. Listing 3.4 shows why this is problematic.

```
1  /**
2   * @param a public
3   * @param b public
4   * @param flag public 1
5   * @param dummy secret 1
6   * @return public
7   */
8  int select (int a, int b, int *flag, int *dummy){
9    int result = *flag ? a : b; // checker thinks flag is secret
10                               // because of the following line
11   flag = dummy; // this should have no effect, but it does
12   return result;
13 }
```

Listing 3.4: Type checker thinks flag is secret and labels the code as insecure

flag points to public data during the entire execution of the program, until it gets overwritten and points to secret data (Line 11). This imposes no threat to the program, because it never gets used afterwards. However, for the type checker it looks like *flag* is secret from the beginning, consequently labeling the program is insecure, because it "branches on secret data" in Line 9.

Lastly, scVerif is not always able to partially evaluate programs, for instance when the control-flow of a program depends on it's parameters and scVerif doesn't know their concrete values.

3.4 Dynamic Leakage Models

TS1 was unable to handle memory accesses. This was solved by checking the execution trace after scVerif partially evaluated the program. However, it led to the checker being unable to detect secret-dependent control-flow, because the entire control-flow gets evaluated during the partial evaluation.

To solve this problem, we can make use of one of scVerif's important properties: it's leakage preserving, meaning it will never remove a leak from the execution trace. Instead of encoding the used leakage models in the type system, we will now encode them explicitly in the hardware model it-

self. This way leakages cannot get removed from the trace anymore. For detecting secret-dependent branching, macros similar to Listing 3.5 are used.

```
1 // branch if not equal
2 macro beqn1_leak (label l)
3 {
4   leak evaluatedGuard(apsrz);
5 }
```

Listing 3.5: PCM modeled using explicit leak statement

This macro encodes secret-dependent control-flow the same way the PCM does it: all evaluated guards are leaked. These guards are saved in a special register, like the zero flag "apsrz" which is used for (in)equality. Line 3 instructs scVerif to leak this register when performing an conditional branch that gets executed when two operands are not equal. The macros for the other branching instructions (equality, less or greater than, etc.) are not shown in this example. They all look similar.

For modeling secret-dependent memory accesses, the macros of Listing 3.6 is used.

```
1  macro ldr3_leak (w32 destination, w32 base, w32 offset)
2    w32 val
3  {
4    leak ldrOperand (base);
5    leak ldrOperand (offset);
6  }
7
8  macro str3_leak (w32 value, w32 base, w32 offset)
9  {
10   leak strOperand (base);
11   leak strOperand (offset);
12 }
```

Listing 3.6: MOM modeled using explicit leak statement

Memory can be accessed for reading (captured by the Lines 1 - 6) and writing (Lines 8 - 12). Both macros follow the definition of MOM, assuming that the address of every memory access is leaked. An address contains two parts: the base position in memory, leaked in Lines 4 and 10, as well as an offset, leaked in Lines 5 and 11.

Using these, some of the imposed restrictions on Section 3.2.2's TS are no longer required, because the leakage behavior is now fully encoded in the hardware model. TS2's rules are only used for taint propagation. The following rules replace their old counterparts:

$$\frac{\Gamma \vdash e : t \qquad x \text{ array}}{\Gamma \vdash x[e] : \tau(x[e])}(array) \tag{3.15}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash < e >: \tau(< e >)}(memory\ access) \tag{3.16}$$

$$\frac{\Gamma \vdash e : t \qquad \Gamma \vdash c_t : t_t \qquad \Gamma \vdash c_f : t_f \qquad t \leq t_t \wedge t \leq t_f}{\Gamma \vdash \textbf{if } e \textbf{ then } c_t \textbf{ else } c_f : max\{t_t, t_f\}}(\textit{if else}) \tag{3.17}$$

$$\frac{\Gamma \vdash e : t \qquad \Gamma \vdash c : t' \qquad t \leq t'}{\Gamma \vdash \textbf{while } e \textbf{ do } c : t'}(\textit{while}) \tag{3.18}$$

The expression e of an memory or array access is no longer required to be public (Rule 3.15 and Rule 3.16). They replace Rule 3.13 and Rule 3.14. The same holds for the expression e of control-flow statements (Rule 3.17 and Rule 3.18). The final TS now allows branching on secrets replacing the rules Rule 3.7 and Rule 3.8.

3.5 Evaluation

TS2 is able to fix most of the problems the previous type systems had. It can detect all of Section 2's listed sources of timing side-channels in executables. In fact it works for every possible leakage model, if it is also explicitly encoded in the hardware model, similar to how we defined the other leakage models. However, it is not able to verify real cryptographic implementations, because important features are still missing:

First, the type checker still operates on the execution trace from the partial evaluation, which might cause false positives as demonstrated in Listing 3.4.

Second, TS2 is still limited to programs that can be partially evaluated. This impedes verifying programs such as "memcpy", "memcmp" and "memset", which are frequently used. These can only be evaluated if the size parameter is fixed and provided to scVerif. In practice, this is not a big problem, because the tool can still perform the analysis for every possible size parameter that might occur separately.

Third, scVerif only allows accessing 32 bit variables. Cryptographic implementations make frequent use of 64 bit variables, as well as 8 bit variables in bit-wise manipulations, so this might cause some problems.

Last, global variables are not supported in this thesis. These are often used for S-Boxes (tables containing precomputed values) to speed up algorithms like AES [11]. Section 4.3 explains how support for global variables can be added.

In order to still verify these implementations, it would be be necessary to rewrite them in a way such that they comply to the restrictions. This defeats the purpose, because the checker would no longer verify the original implementation, but an alternate, specifically crafted version. This alternate version can have entirely different side-channel characteristics. On top of that the compiler might optimize the alternative differently than the original. Therefore both versions have to be treated as different programs and verifying the alternative provides no guarantee for the original.

4 Inferring the Initial Security Types

Languages like Python or Javascript infer the type of variables by themselves, so there is no need to declare the type specifically as in other languages, e.g. "int a = 42;". This is possible, because the type of a variable can usually be inferred from the instructions it is used on. For security types this not possible, because they are not part of most languages themselves and consequently cannot be inferred based on semantics. Another problem is that the set of secret variables is undecidable. Without them the checker would function as a mere leakage detector, detecting which variables are leaked and are therefore not constant-time, because they influence the timing behavior of the program. The interpretation whether a program is constant-time or not would be missing, because the checker doesn't know the secrecy of the leaking variables. In order to get these types and produce a definitive answer, the types need to be specified manually by a human.

For this, three methods were compared for tainting data, where tainting means marking data as secret: using NOP-functions (Section 4.1), compiler attributes (Section 4.2) and source code comments (Section 4.3).

4.1 Using NOP-Functions to Annotate Variables

NOP stands for *no operation*. A NOP-function therefore is a function doing nothing. The idea is to call the function with the data you want to taint as secret and then during the analysis, look for this particular function call and mark it's parameters as secret. This idea is used in [2]. A small example, showcasing how this approach could be used, can be seen in Listing 4.1.

```
1 #include "taint.h"
2
3 int main(){
4     int a = 42; taint(&a, sizeof(a));
5     int b = 17;
6
7     return a ^ b;
8 }
```

Listing 4.1: Annotating variable a as secret using a nop-function

In this example, the variable *a* gets tainted in line 4 by calling *taint* with a pointer to *a*, to get the memory region. The second argument, the size, can be used to specify how much of the memory region is secret. In the example, the entire variable is tainted. If *a* was an array, the size parameter could be used to explicitly taint only the first *n* elements.

Using this method has the advantage that it directly taints the specified memory region, so every pointer to the same region is automatically tainted as well. However, there is also a major downside to

such an invasive approach. It modifies the source code of the program. This could cause the compiler to optimize the program in a different way. The compiler could also optimize the call to the function away, because it does nothing. During testing, this behavior could not be observed, but that does not provide any formal guarantee.

Because of its invasive nature, modifying the source code and being potentially unreliable, this method is inadequate.

4.2 Using Compiler Attributes to Annotate Variables

Clang features a built-in function called *annotate* for adding further information in the form of attributes to the code [1], without the downside of modifying the program semantically. Plugins can use these annotations for their transformations. An example can be seen in Listing 4.2.

```
1 int main(){
2     __attribute__((annotate("secret"))) int a = 42;
3     __attribute__((annotate("public"))) int b = 17;
4
5     return a ^ b;
6 }
```

Listing 4.2: Annotating two variables using Clang's annotate function

Line 2 and 3 describe how clang can be instructed to attribute *a* and *b* with an annotation of "secret" and "public" respectively. This approach allows annotating variables with more than just "secret", in contrast to the previous approach, which could only taint variables as secret and everything else would be public. In theory, this could be used for adding more security types, like "random" or "declassified" for hashed secrets.

Unfortunately, these annotations don't propagate into the binary [36], so they cannot be accessed in verification frameworks independent of the clang compiler.

4.3 Using Source Code Comments to Annotate Variables

The final method, used in this thesis, involves using Doxygen comments (as in every C code example in this thesis). Doxygen is a widely used tool for commenting and annotating source code [34]. Its primary use case is to generate documentation for interfaces by annotating the parameters and the return type of functions. We will use it, to annotate parameters as either "public" or "secret". However, other types, like "random" or "declassified", could be added as well. Out of the box, it is not capable of annotating local or global variables. Fortunately, it can easily be extended using Doxyfiles with custom keywords. This way the additional keywords 'local' and 'global' were added.

Doxygen can output the documentation in XML files. These can then be parsed to extract the initial security types. Mapping from source code annotations to the registers and memory regions used in executables involves little work for function parameters. In ARM-Assembly there is a simple calling convention [7]: r0 contains the first parameter, r1 the second, etc. In the end, r0 is also used for the return value of functions. Utilizing this convention makes the mapping straightforward.

Mapping annotations to the memory locations of local and global variables involves more work. This was not prepared in the scope of this thesis, but following is a brief description of how it could be achieved:

Executables have a so-called *dwarf debug sections*, containing information about all functions and their variables [17]. This can be used to look up where variables are stored in memory. Debuggers use the same sections to look up where variables are stored, which type they have or where they were declared. Listing 4.3 is a short extract of the debug section generated for Listing 4.2.

```
1  <1><2d>: Abbrev Number: 2 (DW_TAG_subprogram)
2     <2e>   DW_AT_external    : 1
3     <2e>   DW_AT_name        : (indirect string, offset: 0x3f): main
4     <32>   DW_AT_decl_file   : 1
5     <33>   DW_AT_decl_line   : 1
6     <34>   DW_AT_type        : <0x67>
7     <38>   DW_AT_low_pc      : 0x660
8     <40>   DW_AT_high_pc     : 0x1a
9     <48>   DW_AT_frame_base  : 1 byte block: 9c         (DW_OP_call_frame_cfa)
10    <4a>   DW_AT_GNU_all_call_sites: 1
11    <4a>   DW_AT_sibling     : <0x67>
12 <2><4e>: Abbrev Number: 3 (DW_TAG_variable)
13    <4f>   DW_AT_name        : a
14    <51>   DW_AT_decl_file   : 1
15    <52>   DW_AT_decl_line   : 2
16    <53>   DW_AT_type        : <0x67>
17    <57>   DW_AT_location    : 2 byte block: 91 6c      (DW_OP_fbreg: -20)
18 <2><5a>: Abbrev Number: 3 (DW_TAG_variable)
19    <5b>   DW_AT_name        : b
20    <5d>   DW_AT_decl_file   : 1
21    <5e>   DW_AT_decl_line   : 3
22    <5f>   DW_AT_type        : <0x67>
23    <63>   DW_AT_location    : 2 byte block: 91 68      (DW_OP_fbreg: -24)
```

Listing 4.3: Extract of the .debug_info section of Listing 4.2

It shows that the function *main* (line 1 - 11) has two local variables *a* (line 12 - 17) and *b* (18 - 23). The memory location of a variable is stored in the attribute *DW_AT_location*. *a* is located at the function's stack frame with an offset of -20 (line 17). *b* can be found in the same stack frame with an offset of -24 (line 23). A similar approach can be used for global variables as well.

The entire workflow for verifying executables now looks the following:

1. User annotates program with security annotations
2. The program gets compiled
3. Doxygen generates the documentation of the program
4. A program called "doxy2il" then generates an IL file from the documentation

5. scVerif is called on the IL file and performs the analysis

The last four steps can be fully automated so users only have to annotate the programs they want to check. There are also different logging levels. These can be used to check which registers are leaking secret data during the execution, if the program is not constant-time. A mapping back to the original source code is not implemented, but highly desirable for better usability.

Using this workflow could prevent regression bugs similar to the Lucky 13 side-channel vulnerability in OpenSSL (Section 2.2), by adding it to the tests in existing CI/CD pipelines, once real cryptographic implementations are supported by the proposed method.

5 Conclusion

Existing formal tools for verifying constant-time implementations analyze implementations either at the source code level or during the compilation. They ignore that the compiler might optimize software-based countermeasures away. This thesis therefore proposes a novel, formal method for automatically verifying constant-time implementations of executables for concrete hardware architectures. Doing so verifies that the final programing is resilient against cache-timing attacks. The approach consists of adding a type checker for an information-flow type system for scVerif's intermediate language (IL). IL features an explicit leak statement, which can be used to model the entire (cache-timing) side-channel behavior of an ISA. The type system uses these explicit leak statements instead of following particular leakage models. Consequently, it is able to detect timing side-channels for *arbitrary leakage models*. This makes the method extremely flexible and distinct to existing tools which only consider fixed, hardcoded leakage models. Furthermore, it also enables considering timing side-channels caused by *variable-timing hardware instructions*. These are usually not considered by other formal tools at all. The type checker cannot infer the initial security types of every variable. Human-generated doxygen comments are used for providing this information.

Incorporating this method into existing projects requires little work, as users only have to provide easy to use security annotations. The entire verification can be automated and therefore added into existing CI/CD pipelines to prevent regression bugs causing new side-channel vulnerabilities. This can prevent incidents like OpenSSL's Lucky 13 side-channel vulnerability from occurring.

However, the current implementation causes the type checker to only work with restricted programs. The static type checker cannot deal with pointers itself. Instead of writing a points-to analysis to cope with this problem, we rely on scVerifs ability to partially evaluate programs, which rewrites every memory access to variable accesses and then operates on the resulting execution trace. The partial evaluating doesn't work on every program. Every function using parameters in control-flow statements cannot be partially evaluated, if scVerif doesn't know the concrete value of these. This already prevents checking programs like "memcpy" or "memcmp", for any parameter. Yet the approach works for partially fixed parameters, i.e. specific number of bytes to copy but symbolic memory values.

Using the execution trace also causes the type checker to generate some false positives for certain artificial cases. These should not occur in real cryptographic implementations. This behavior could be prevented by implementing the type checker as part of the partial evaluation.

5.1 Future Work

The proposed method has only been tested for trivial programs, like the listings presented in the previous sections. Testing it on more sophisticated, cryptographic implementations is needed. However, before this can be done, some improvements have to be made:

First, the two previously mentioned problems have to be fixed. Implementing a points-to analysis seems reasonable, because it fixes both of them. The type checker would no longer depend on scVerifs partial evaluation and therefore be able to check an increased number of programs.

Second, scVerif currently only supports accessing variables with a word size of 32 bits. This is not sufficient for every cryptographic implementation. For example, variables with a word size of 8 bits are often used for extracting a byte of larger variables. Fixing this would increase the number of programs the type checker can analyze even further.

Third, global variables have to be supported, as S-Boxes (lookup tables) that are often used to speedup algorithms like AES, are usually implemented as global variables. Section 4.3 explains how this can be achieved.

Fourth, there is no way of telling the type checker that a variable gets *declassified* (is no longer secret) by a function. It would be a great improvement if declassification would be added. This is important, as leaking the hash of secret data imposes no threat to the program. But the hashing functions cache-timing behavior still needs to be verified to be constant-time.

Last, the tool only tells users what registers are leaking secret data during which kind of instruction. There is no mapping back to the original source code. Adding this optional feature would greatly improve usability, as users would not have to check every single occurrence of that leaking instruction to find the correct place in the original source code.

List of Abbreviations

ISA instruction set architecture

TS type system

IFC information-flow control

PCM program counter model

MOM memory obliviousness model

CTP constant-time policy

CTC constant-time with cost model

COM cost obliviousness model

List of Figures

Listings

Bibliography

[1] *Clang: a c language family frontend for llvm*. https://clang.llvm.org/.

[2] *ctgrind*. https://github.com/agl/ctgrind/. Accessed: 2020-05-26.

[3] *tis-ct*. https://trust-in-soft.com/tis-ct/.

[4] N. J. AL FARDAN AND K. G. PATERSON, *Lucky thirteen: Breaking the tls and dtls record protocols*, in 2013 IEEE Symposium on Security and Privacy, IEEE, 2013, pp. 526–540.

[5] J. B. ALMEIDA, M. BARBOSA, G. BARTHE, F. DUPRESSOIR, AND M. EMMI, *Verifying constant-time implementations*, in 25th USENIX Security Symposium (USENIX Security 16), 2016, pp. 53–70.

[6] J. B. ALMEIDA, M. BARBOSA, J. S. PINTO, AND B. VIEIRA, *Formal verification of side-channel countermeasures using self-composition*, Science of Computer Programming, 78 (2013), pp. 796–812.

[7] ARM, *Procedure call standard for the arm® 64-bit architecture (aarch64)*. https://github.com/ARM-software/abi-aa/blob/master/aapcs64/aapcs64.rst#611general-purpose-registers.

[8] G. BARTHE, G. BETARTE, J. CAMPO, C. LUNA, AND D. PICHARDIE, *System-level non-interference for constant-time cryptography (full version)*, (2014).

[9] G. BARTHE, M. GOURJON, B. GRÉGOIRE, M. ORLT, C. PAGLIALONGA, AND L. PORTH, *Masking in fine-grained leakage models: Construction, implementation and verification*. Cryptology ePrint Archive, Report 2020/603, 2020. https://eprint.iacr.org/2020/603.

[10] G. BARTHE, B. GRÉGOIRE, AND V. LAPORTE, *Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time"*, in 2018 IEEE 31st Computer Security Foundations Symposium (CSF), IEEE, 2018, pp. 328–343.

[11] D. J. BERNSTEIN, *Cache-timing attacks on aes*, (2005).

[12] S. BLAZY, D. PICHARDIE, AND A. TRIEU, *Verifying Constant-Time Implementations by Abstract Interpretation*, in European Symposium on Research in Computer Security, 22nd European Symposium on Research in Computer Security, Oslo, Norway, Sept. 2017.

[13] D. BRUMLEY AND D. BONEH, *Remote timing attacks are practical*, Computer Networks, 48 (2005), pp. 701–716.

[14] S. CAULIGI, G. SOELLER, F. BROWN, B. JOHANNESMEYER, Y. HUANG, R. JHALA, AND D. STEFAN, *Fact: A flexible, constant-time programming language*, in 2017 IEEE Cybersecurity Development (SecDev), IEEE, 2017, pp. 69–76.

[15] S. A. CROSBY, D. S. WALLACH, AND R. H. RIEDI, *Opportunities and limits of remote timing attacks*, ACM Transactions on Information and System Security (TISSEC), 12 (2009), pp. 1–29.

[16] W. DE GROOT, *A Performance Study of X25519 on Cortex-M3 and M4*, PhD thesis, Eindhoven University of Technology, 2015.

[17] M. J. EAGER, *Introduction to the dwarf debugging format*, 2012.

[18] D. GOLLMANN, *Computer security*, Wiley, third ed., 2011.

[19] INTEL, *Intel 64 and ia-32 architectures optimization reference manual*, Intel Corporation, Sept, (2014).

[20] G. IRAZOQUI, M. S. INCI, T. EISENBARTH, AND B. SUNAR, *Lucky 13 strikes back*, in Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, 2015, pp. 85–96.

[21] P. C. KOCHER, *Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems*, in Annual International Cryptology Conference, Springer, 1996, pp. 104–113.

[22] C. LATTNER AND V. ADVE, *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, in Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar 2004.

[23] X. LEROY ET AL., *The compcert verified compiler*, Documentation and user's manual. INRIA Paris-Rocquencourt, 53 (2012).

[24] D. LEVINTHAL, *Performance analysis guide for intel core i7 processor and intel xeon 5500 processors*, Intel Performance Analysis Guide, 30 (2009), p. 18.

[25] B. MOLLER, *Security of cbc ciphersuites in ssl/tls: Problems and countermeasures*, http://www.openssl.org/~bodo/tls-cbc.txt, (2004).

[26] K. MOWERY, S. KEELVEEDHI, AND H. SHACHAM, *Are aes x86 cache timing attacks still feasible?*, in Proceedings of the 2012 ACM Workshop on Cloud computing security workshop, 2012, pp. 19–24.

[27] D. A. OSVIK, A. SHAMIR, AND E. TROMER, *Cache attacks and countermeasures: the case of aes*, in CryptographersâĂŹ track at the RSA conference, Springer, 2006, pp. 1–20.

[28] M. PEINADO AND T. KIM, *System and method for providing stealth memory*, Mar. 31 2015. US Patent 8,996,814.

[29] O. REPARAZ, J. BALASCH, AND I. VERBAUWHEDE, *Dude, is my code constant time?*, in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, IEEE, 2017, pp. 1697–1702.

[30] B. RODRIGUES, F. M. QUINTÃO PEREIRA, AND D. F. ARANHA, *Sparse representation of implicit flows with applications to side-channel detection*, in Proceedings of the 25th International Conference on Compiler Construction, 2016, pp. 110–120.

[31] R. SISON AND T. C. MURRAY, *Verifying that a compiler preserves concurrent value-dependent information-flow security*, CoRR, abs/1907.00713 (2019).

[32] J. SOMOROVSKY, *Curious padding oracle in openssl (cve-2016-2107)*, On Web-Security and-Insecurity blog, (2016). `https://web-in-security.blogspot.com/2016/05/curious-padding-oracle-in-openssl-cve.html`.

[33] B. STEENSGAARD, *Points-to analysis in almost linear time*, in Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1996, pp. 32–41.

[34] D. VAN HEESCH, *Doxygen: Source code documentation generator tool*. `https://www.doxygen.nl`.

[35] D. VOLPANO, C. IRVINE, AND G. SMITH, *A sound type system for secure flow analysis*, Journal of computer security, 4 (1996), pp. 167–187.

[36] S. T. VU, K. HEYDEMANN, A. DE GRANDMAISON, AND A. COHEN, *Compilation and optimization with security annotations*, 2019.

[37] Y. YAROM AND K. FALKNER, *Flush+ reload: a high resolution, low noise, l3 cache side-channel attack*, in 23rd USENIX Security Symposium (USENIX Security 14), 2014, pp. 719–732.

[38] Y. YAROM, D. GENKIN, AND N. HENINGER, *Cachebleed: a timing attack on openssl constant-time rsa*, Journal of Cryptographic Engineering, 7 (2017), pp. 99–112.