UC Irvine

UC Irvine Electronic Theses and Dissertations

Title

Soteria: Automatically Mitigating Timing Side-Channel Vulnerabilities in Sensitive Programs

Permalink

https://escholarship.org/uc/item/0fn3s0c7

Author

Unal, Arda

Publication Date

2022

Peer reviewed|Thesis/dissertation

eScholarship.org

Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

SOTERIA: Automatically Mitigating Timing Side-Channel Vulnerabilities in Sensitive Programs

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Arda Ünal

Thesis Committee:
Assistant Professor Joshua Garcia, Chair
Associate Professor James Jones
Professor Sam Malek

2022

© 2022 Arda Ünal

TABLE OF CONTENTS

Page

LIST OF FIGURES **iv**

LIST OF TABLES **v**

LIST OF ALGORITHMS **vi**

LIST OF LISTINGS **vii**

ACKNOWLEDGMENTS **viii**

ABSTRACT OF THE THESIS **ix**

1 Introduction **1**

2 Background **6**

3 Soteria **11**
- 3.1 Identifying and Transforming Side-Channel Inducing Patterns 13
  - 3.1.1 Early Return Rewrite Transformation 13
  - 3.1.2 Secret-Dependent Assignment Rewrite Transformation 15
  - 3.1.3 Final Branchless Transformation 18

4 Evaluation **20**
- 4.1 RQ1: Correctness of Converted Side-Channel Benchmarks 21
- 4.2 Updates to Fix Errors in Original Benchmarks 22
- 4.3 RQ2: Constant-Time Effectiveness of Transformations 24
- 4.4 RQ3: Functional Correctness of Transformed IR 26

5 Threats to Validity **29**
- 5.1 Construct Validity 29
- 5.2 Internal Threat to Validity 30
- 5.3 External Threat to Validity 31

6 Related Work **32**

7 Conclusions and Future Work **35**

Bibliography **37**

LIST OF FIGURES

Page

3.1 Overview of Soteria 12
3.2 Overview of Soteria's transformations 13

LIST OF TABLES

Page

3.1 Patterns used by Soteria to identify side-channel inducing code 12

4.1 Correctness Results for Rewritten Benchmarks 21
4.2 The results of differential fuzzing on converted benchmarks' constant-time behavior. Red cells indicate that they could not be compared since automatically mitigated and updated, safe versions are not available in the original Java benchmarks. 27
4.3 Results of the Running Example 28

LIST OF ALGORITHMS

		Page
1	Matcher used to find branching on secret	13
2	Transformation rule to rewrite early termination	14
3	Matcher used to find secret-dependent early termination in a loop	15
4	Transformation rule to rewrite early termination in a loop	16
5	Matcher used to find the assignment statements in secret-dependent branches	16
6	Transformation rule to rewrite assignments in secret-dependent branches	17
7	Matcher used to find cmov calls in secret-dependent branches	18
8	Transformation rule to eliminate secret-dependent branches	19

LIST OF LISTINGS

1.1 Unsafe password check . . . 2
2.1 Mitigation attempt susceptible to cache timing attacks . . . 7
2.2 Constant-time select with bitmasks . . . 8
2.3 Nonconstant-time select with branching . . . 9
2.4 Constant-time select with cmov . . . 9
2.5 Constant-time password check . . . 10
3.1 After automatic elimination of early termination . . . 17
3.2 After rewriting secret-dependent assignments using the cmov construct . . . 18
3.3 After eliminating secret-dependent branches . . . 19
4.1 Nonconstant-time array_safe . . . 23
4.2 Updated constant-time array_safe . . . 23
4.3 Nonconstant-time loopandbranch. The red operator is unsafe code removed in the original version; the green operator is the code added that attempted to make the code safe but failed to do so. . . . 24
5.1 Straightline_unsafe benchmark has a fixed number of instructions difference between the imbalanced branches. . . . 30

ACKNOWLEDGMENTS

I want to thank my committee members, Josh Garcia, Jim Jones, and Sam Malek, who have given time and provided critical feedback that has made my dissertation stronger at each milestone.

ABSTRACT OF THE THESIS

Soteria: Automatically Mitigating Timing Side-Channel Vulnerabilities in Sensitive Programs

By

Arda Ünal

Master of Science in Software Engineering

University of California, Irvine, 2022

Assistant Professor Joshua Garcia, Chair

Side-channel attacks try to gain information about the secret data in sensitive programs through leveraging the difference between the algorithm and its implementation. Most common side-channel vulnerabilities arise from timing variations in program execution, memory access patterns, memory, power, and network consumption, response size, electromagnetic emissions, and acoustics that could be tied back to secret information. For these reasons, sensitive programs (e.g., real-world cryptographic code) are written in a constant-time fashion to avoid timing side-channel vulnerabilities. In this thesis, we present an approach that automatically mitigates timing side-channel vulnerabilities through a set of source-to-source transformations resulting in secret-dependent branch-free code so that there will be no secret-dependent timing variation during the execution of the program.

Chapter 1

Introduction

Side-channel vulnerabilities stem from the gap between the abstract machine and its implementation targeting a real system. Side-channel attacks exploit these unintended vulnerabilities caused by the implementation to uncover the secret data from sensitive programs. Side-channel attacks extend from power consumption analysis to exploiting micro-architectural features and optimizations. Kocher [32] exploited the power usage between successive runs of tamper resistant devices such as smart cards and this differential power analysis (DPA) attack is followed by the analysis of electromagnetic emissions of these security devices [43]. We have witnessed the evolution of these attacks over the years. With the rise of cloud computing, side-channel attacks shifted scope to timing attacks that could be performed remotely [17, 12]. These attacks are followed by cache-timing attacks [2, 3, 45, 1, 35, 26, 28] in cloud settings and even on mobile devices [33]. More recently we saw two important attacks that exploited the micro-architectural side-channel vulnerabilities in branch prediction and speculative executions in both the Spectre [30] and Meltdown [34] attacks.

Timing side-channel attacks differ from the other side-channel attacks as they do not require vicinity to the victim machine and can be conducted remotely. They have been demon-

strated in practice over many years with extensive research in both academia and industry: Kocher [31] showed that timing attacks are possible on implementations of cryptographic algorithms (e.g., finding fixed Diffie-Hellman exponents or factor RSA keys); side-channel vulnerabilities helped attackers to steal private keys and caused catastrophic damage in DSA signing operations [24]; and researchers successfully attacked Amazon's TLS implementation using a timing attack [4].

A simple function in Listing 1.1 contains two exploitable timing side-channel vulnerabilities. In this function, there are two vector inputs: `pw`, the password the attack is trying to identify, and `guess`, an attacker-supplied string that is compared against the password. The first side-channel vulnerability arises from an early termination that occurs if the size of `guess` and `pw` are not the same (lines 2-3). An attacker can leverage this vulnerability by trying different lengths for `guess` until the execution time increases. For example, when the secret `pw` $= 123$, execution takes 41 instructions with `guess` $\in \{1, 12, 1234, 12345\}$ and it takes 70 instructions when `guess` $= 456$.

For the second side-channel vulnerability in Listing 1.1, another early termination can occur if elements of `guess` and `pw` at the same index are not equal, leading to another variance in execution time that can be exploited by an attacker (lines 5-6). For example, when the secret $pw = 123$, execution takes 70 instructions when $guess = 456$, terminating after comparing 1 to 4; 99 instructions when $guess = 193$, terminating after comparing 2 to 9; 128 instructions when $guess = 129$, terminating after comparing 3 to 9; and 142 instructions when $guess = 123$, completing the loop without early termination.

```
1 bool pwcheck(std::vector<byte_t> &guess, std::vector<byte_t> &pw) {
2   if (guess.size() != pw.size()) {
3     return false; }
4   for (size_t i = 0; i < guess.size(); i++) {
5     if (guess[i] != pw[i]) {
```

```
6           return false; }}
7   return true;}
```

Listing 1.1: Unsafe password check

Software-based countermeasures are taken to implement constant-time systems using constant time constructs. These constant-time constructs often deviate from conventional programming practices in that they avoid common optimization techniques such as early function and loop terminations. These common optimization techniques can result in discrepancies in execution time and memory access patterns which, in turn, can be used by an attacker to determine if an input was close to matching the secret.

Constant-time implementations of sensitive programs such as cryptographic algorithms avoid leaking secret data through timing attacks by making sure that the variation in the execution time does not depend on the secret data. However, major challenges arise when creating such constant-time code as such code may be optimized away into nonconstant time intermediate representations by compiler optimizations that constant-time code does not account for. Writing constant-time code is not straightforward. Rather, it requires compiler knowledge to understand that a compiler may optimize these constant-time constructs in a way that is likely to deviate from the programmer's intentions. As a result, these deviations may result in nonconstant time intermediate representations and, subsequently, become nonconstant time executables. These vulnerabilities in security-critical software systems escape diligent testing and code-review processes [4, 40, 39, 41] and suggest that (1) more rigorous analysis techniques are required and (2) these constant-time transformations should be done in a systematic way.

Support for side-channel resistance from a mainstream compiler would enable every programmer writing code in the compiler's supported languages—not only side-channel experts such as cryptographers—to secure their code in a systematic way. To address these afore-

mentioned challenges, this thesis makes the following contributions:

- We present Soteria, a compiler-based approach to rewrite sensitive programs that are written in a conventional manner into timing side-channel resistant programs. Soteria provides annotations to mark secret variables in sensitive programs and apply a series of general source-to-source transformations to rewrite these sensitive programs into constant-time counterparts without forcing the programmer to deviate from conventional programming practices. Soteria first matches against code patterns that result in timing side-channel vulnerabilities. It then transforms these matched code patterns using grammar-based rules to eliminate their underlying nonconstant-time behavior.
- We implement Soteria using LibAST Matchers [36] and clang [37] to target C/C++ programs. We focus on C/C++ since the majority of high-value target programs susceptible to timing side channels, such as cryptographic libraries, are written in C/C++—avoiding just-in-time (JIT) compilation (e.g., in Java) that might introduce timing side-channel vulnerabilities [16].
- We evaluate Soteria on C/C++ benchmarks converted from Java benchmarks utilized in the evaluation of the side-channel analysis approaches DifFuzz [38], Blazer [7], and Themis [20].

For the majority of the benchmarks, we found that running programs rewritten by Soteria finds zero or small discrepancies in the number of instructions retired—effectively demonstrating that Soteria eliminates timing side-channel vulnerabilities for the benchmarks. Instructions are retired when they leave the retirement unit—which is a unit of a CPU microarchitecture that knows how and when to commit (i.e., "retire") temporary, speculatively-executed results to permanent architectural state. In other words, the unit will write the results of speculatively executed instructions as if it was executed in-order

when instructions are retired. Using the number of instructions retired over the number of instructions is a more accurate representation of how many instructions were really executed [23].

Chapter 2

Background

The current research for detecting and mitigating timing side-channel vulnerabilities focuses on secret keys and cryptographic algorithms since they are high-value targets [27, 18]. However, it is not sufficient to only harden the cryptographic algorithm implementations to make a security critical system timing side-channel resistant. Ideally, all software in security critical systems, not just cryptographic code, should have resistance against timing side channels.

With SOTERIA, we propose a solution to show that when these solutions are integrated into a mainstream compiler, side-channel resistance can be achieved systematically in a general-purpose programming language without having to design or use a domain-specific language (DSL) as previously done by `qhasm` [13], `Vale` [15], `Jasmin` [5] and `FaCT` [18].

In constant-time implementations, the variation in the execution time or memory access patterns cannot be correlated with secret data. Constant-time implementations take advantage of constant-time constructs such as avoiding memory access patterns that depend on the secret data and branchings that are controlled by secret data. Secret-dependent memory access might leak the address of the element if a cache miss occurs. A cache miss takes a longer time introducing a timing variance that depends on the secret data which could be

exploited by the attacker [45, 1, 35, 26, 28]. If a conditional branching depends on secret data, both the statements and their execution time depend on the secret data introducing a timing side-channel vulnerability.

These timing side channels might be mitigated by adding dummy instructions to each branch to make them identical in terms of the number of instructions executed. However, this mitigation could still be susceptible to memory access-pattern attacks as shown in Listing 2.1, with the mitigation code highlighted in green. The mitigation attempts to balance the number of instructions between the branches by adding a dummy loop from the true branch to the false branch. However, the lookup by index `i` in line 14 of Listing 2.1 might result in a cache miss if the dummy loop evicts the cache line where `a[i]` resides. This cache miss might leak to the attacker which branch is taken.

```
1 int array_unsafe(PUBLIC std::vector<int> &a, SECRET int taint) {
2   int t = 0;
3   if (taint < 0) {
4     int i = a.size()-1;
5     while (i >= 0) {
6       t = a[i];
7       i--;}}
8   else {
9     int i = 0;
10    t = a[i] / 2;
11    i = a.size();}
12  return t;}
```

Listing 2.1: Mitigation attempt susceptible to cache timing attacks

For this reason, eliminating these conditional branches by writing them in a flat manner using bitmasks would be a more robust solution as shown in Listing 2.2. In this `select` function, `isnonzero` variable gets assigned to 1 when the predicate is true and 0 otherwise. Consequently, variable `mask` gets assigned to 0xff...ff when the predicate is true and 0 otherwise.

When the mask is 0xff...ff, variable `ret` becomes $old_val \oplus new_val \oplus old_val$ ends up being equal to new_val whereas when the mask is zero `ret` becomes $0 \oplus old_val$ which is equal to old_val. Consequently, eliminating the branch-dependent assignments using bitmasks would result in a uniform memory access pattern.

```
1 /* Conditionally return new_val or old_val depending on whether pred is
     set */
2 /* Semantically equivalent to:
3     return pred ? new_val : old_val */
4 unsigned select (unsigned pred, unsigned old_val,
5                                  unsigned new_val) {
6     unsigned isnonzero = (pred | -pred) >>
7                       (sizeof(unsigned) * CHAR_BIT - 1);
8     /* -0 = 0, -1 = 0xff....ff */
9     unsigned mask = -isnonzero;
10    unsigned ret = mask & (old_val^new_val);
11    ret = ret ^ old_val;
12    return ret;}
```

Listing 2.2: Constant-time select with bitmasks

In some popular hardware architectures, there are conditional move operations that could be utilized to eliminate these branches rather than using bitmasks, such as the CMOVcc instruction in x86 [21]. These kinds of conditional move operations enable assignment operations to be done without branch instructions which are subject to speculative execution (i.e., executing code before it is known the code is needed). With the use of the CMOV instruction, such execution is avoided because such operations are implemented as ALU select operations—which are a type of operation that is not subject to predictive or speculative execution—as opposed to branch instructions—which are the current types of instructions leveraged by state-of-the-art compiler-based side-channel resistance techniques [42].

These conditional move operations enable us to remove the secret-dependent branches while

keeping the behavior of these branches intact. For example, the following branching example in Listing 2.3 could be replaced by its branchless `cmov` counterpart in Listing 2.4. The `branch` function takes 38 instructions when the branch in line 4 of Listing 2.3 is taken and 40 instructions when it is not, whereas the `cmov_select` function in Listing 2.4 always takes 39 instructions since it has no branching.

```
1 unsigned branch(unsigned pred, unsigned old_val, unsigned new_val) {
2   unsigned ret = old_val;
3   if(pred == 1)
4     ret = new_val;
5   return ret;}
```

Listing 2.3: Nonconstant-time select with branching

```
1 unsigned cmov_select(unsigned pred, unsigned old_val, unsigned new_val)
    {
2   unsigned ret = 0;
3   __asm__ __volatile__(
4       "mov %3, %0;"
5       "test %1, %1;"
6       "cmovz %2, %0;"
7       : "=&r"(ret)
8       : "r"(pred), "r"(old_val), "r"(new_val));
9   return ret;}
```

Listing 2.4: Constant-time select with cmov

There are other important details to take into consideration such as nonconstant-time library functions; for example, if the `size()` function were not constant-time and if it is called on the secret data, the length of the secret data would be leaked. Additionally, one should be careful with short-circuit evaluation which will terminate the evaluation of the boolean expression without evaluating it fully if the result of this expression can be determined by the evaluation of its first argument. For example, the expression $b_1 \wedge b_2$ is already known to be false if $b_1 = false$ without having to evaluate b_2. This short-circuit evaluation might leak information about the secret data when there is a time-consuming predicate.

Returning to our unsafe password check example in Listing 1.1, comparing two buffers of arbitrary length is not straightforward to implement in a constant-time fashion and is susceptible to memory access-pattern (i.e., cache timing) attacks. For this reason, there should be an assumption or an agreement as to a maximum length allowed and the execution time of this function should only depend on the properties of an attacker-controlled `guess`. As a result, for this example, we assume that an attacker-controlled `guess` will not be longer than the secret `pw`. The example code snippet shown below in Listing 2.5 is a version of the `pwcheck` function without the timing side-channel vulnerabilities in Listing 1.1. This `pwcheck` function does not perform an early termination when the size of the `guess` and `pw` buffers do not match or when the elements that are in the same index location are not equal to each other, resulting in a scan of the entire `guess` buffer for every execution. Furthermore, memory access patterns become identical for every execution hardening the program against the cache timing attacks. Since the loop always iterates over the whole `guess` buffer, execution time of this program will only depend on the length of the attacker-controlled `guess` buffer which is already known to the attacker. Consequently, these countermeasures harden the program in Listing 1.1 against aforementioned side-channel vulnerabilities.

```
1 #define SECRET [[clang::annotate("secret")]]
2
3 bool pwcheck(std::vector<byte_t> &guess, SECRET std::vector<byte_t> &pw
    ) {
4   bool result = true;
5   for (size_t i = 0; i < guess.size(); ++i) {
6       result &= guess[i] == pw[i];}
7   return result & (guess.size() == pw.size());}
```

Listing 2.5: Constant-time password check

Chapter 3

Soteria

Our approach, SOTERIA, as depicted in Figure 3.1, allows a programmer to annotate the secret and public values in C/C++ source code and automatically mitigates the timing side-channel vulnerabilities through a set of source-to-source transformations. SOTERIA produces flat code that will no longer have secret-dependent timing variation during the execution of the program. SOTERIA is implemented using LibAST Matchers and clang to target C/C++ programs. The LibAST Matcher library in LLVM makes it easy to match AST nodes using an embedded domain-specific language as a query language. Using this language enables us to find specific AST nodes with complex predicates and rewrite them in a constant-time way.

Certain common coding patterns in conventional programming practices might lead to timing side-channel vulnerabilities if they are secret-dependent, such as early termination to minimize execution time. We explain these patterns, the manner in which they can reveal secrets, and the manner in which they can be rewritten in a constant-time fashion using the password comparison code in Listing 1.1 as a running example. To specify code patterns that induce side-channel vulnerabilities, we utilize the patterns described in Table 3.1.

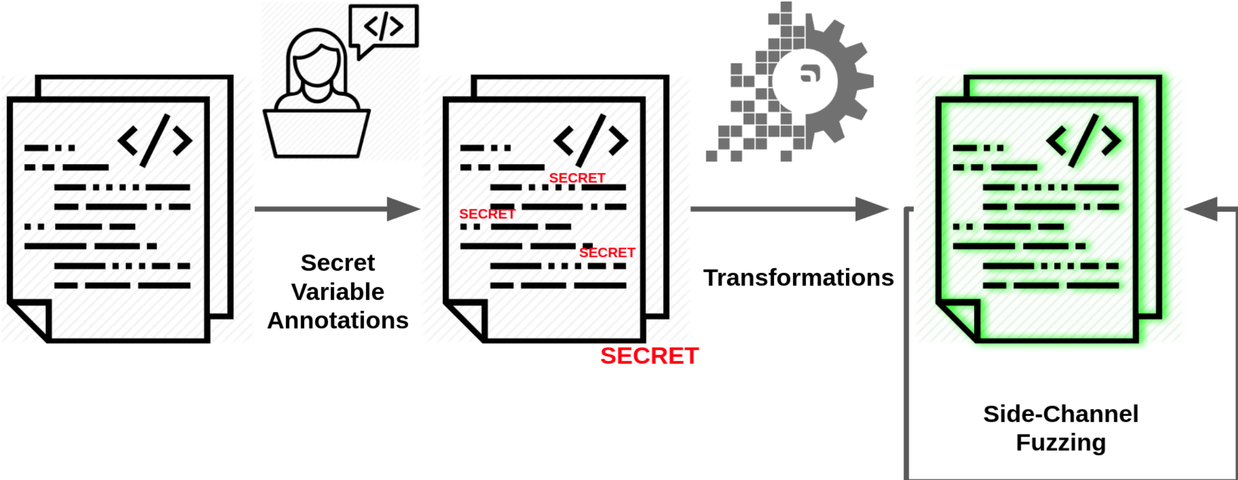

Figure 3.1: Overview of SOTERIA

Each row of Table 3.1 describes the form of each pattern that consists of code statements or expressions. SOTERIA matches and transforms (1) code patterns resulting in early terminations of a function that is dependent on secret information, (2) assignment statements dependent on secret information, and (3) any remaining secret-dependent branches in the code to be flat. Figure 3.2 depicts these transformations as a workflow. Together, identifying and transforming these three types of code patterns achieve side-channel resistance.

Code Patterns	**Description**
$f(x:T)\{\,Stmt\,\}:T_{ret}$	Function declaration with argument x of type T and return value of type T_{ret}
$for(expr_{init}, expr_{cond}, expr_{inc})$ $\{\,Stmt_{body}\,\}$	For statement
$while(expr_{cond})$ $\{\,Stmt_{body}\,\}$	While statement
$expr_{secret}$	Expression to a secretly annotated declaration
$ifStmt(expr)$ $\{\,Stmt_{if}\,\}$ $else$ $\{\,Stmt_{else}\,\}$	If and Else conditional branching
$return$	Return Statement
$expr_{LHS} \leftarrow expr_{RHS}$	Assignment Statement
$cmov(expr, expr_{old}, expr_{new})$	cmov Statement

Table 3.1: Patterns used by SOTERIA to identify side-channel inducing code

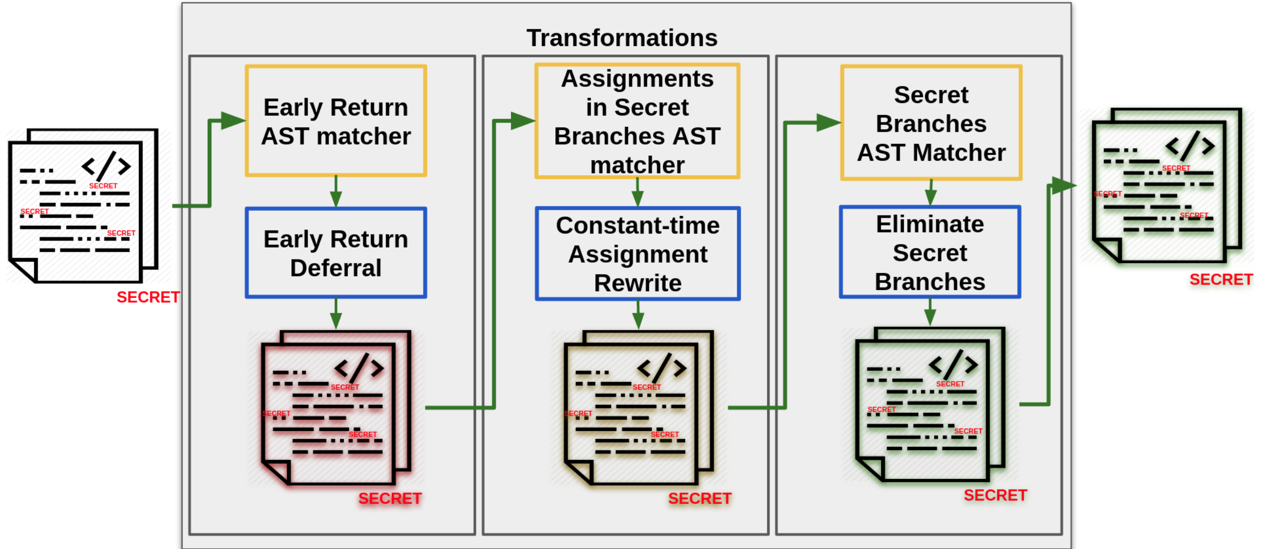

Figure 3.2: Overview of SOTERIA's transformations

3.1 Identifying and Transforming Side-Channel Inducing Patterns

3.1.1 Early Return Rewrite Transformation

As mentioned in the previous section, secret-dependent branches with an early return statement need to be eliminated in order to avoid secret-dependent variance in execution time. First, we need to identify the secret-dependent branches that have a return statement. To that end, the formalized matcher in Algorithm 1 matches AST nodes that have a conditional statement branching on secret ($expr_{secret}$) and have a return statement on either branch of the conditional. For example, in Listing 1.1, the conditional statement in line 2 will match.

Algorithm 1 Matcher used to find branching on secret

$$\exists ifStmt(expr_{secret}) :$$
$$(returnStmt \in Stmt_{if} \vee returnStmt \in Stmt_{else})$$

After matching these AST nodes, the transformation rule in Algorithm 2 will be used to

Algorithm 2 Transformation rule to rewrite early termination

Before

```
1: function F(x : T): T_ret
2:     Stmt_1
3:     if expr_secret then
4:         return expr_ret        ▷ Early-terminating secret-dependent branch
5:     end if
6:     Stmt_2
7:     return expr_retFin
8: end function
```

After

```
1: function F(x : T): T_ret
2:     T_ret return_val ← expr_retFin
3:     SECRET bool not_return ← true
4:     Stmt_1
5:     if expr_secret ∧ not_return then
6:         return_val ← expr_ret
7:         not_return ← false
8:     end if
9:     if not_return then
10:        Stmt_2
11:    end if
12:    return return_val
13: end function
```

rewrite these matched nodes. The return statement in the matched branch will be replaced by $return_val = expr_{ret}$ statement. To accommodate this change, initialization of this variable is added to the beginning of the function assigning $expr_{retFin}$, which is the default return value, to the newly introduced variable $return_val$. To keep the behavior of the program the same, statements that come after this change need to be wrapped inside a branch that will be taken if and only if the function does not terminate early at this stage. For this reason, the boolean variable not_return is introduced and $Stmt_2$ is wrapped in a branch conditioned on this variable. Finally the default return statement is rewritten to return $return_val$.

After automatically eliminating the return statement in the if statement in line 3 of Listing 1.1, we get the following code snippet in lines 3, 5-9, and 16 in Listing 3.1 shown in color brown ■. Similar to the previous rule, early termination of the loop also needs to be eliminated. The matcher in Algorithm 3 is utilized to match every `for` loop with a body that has a secret-dependent branch with a return statement. For our running example in Listing

1.1, lines 4-6 will be matched.

Algorithm 3 Matcher used to find secret-dependent early termination in a loop

$$\forall for(Stmt_{init}, expr_{cond}) \ni \{\, Stmt_{body} \,\} \ni$$
$$(ifStmt(expr_{secret}) \wedge$$
$$(returnStmt \in Stmt_{if} \vee returnStmt \in Stmt_{else}))$$

The transformation in Algorithm 4 is very similar to the previous one in Algorithm 2. However, unlike the previous rule, statements both before (as represented by $Stmt_1$) and after (as represented by $Stmt_2$) this early-terminating if branch are wrapped in two new branches (i.e., lines 5-6 and lines 12-13 of the After section of Figure 4). Wrapping statements before and after the early-terminating if branch ensures that the code keeps iterating until the loop naturally terminates. To keep the behavior of the loop the same, statements both before and after the secret-dependent branch are wrapped in new branches as they would not be executed during iterations after a premature termination. After automatically eliminating the return statement in the if statement in lines 5-6 of Listing 1.1, we get the following code snippet in lines 11-15 in Listing 3.1 shown in color green ■.

3.1.2 Secret-Dependent Assignment Rewrite Transformation

The next step in the sequence of transformations takes the output of the previous section (i.e., source code that is free of early terminations) and prepares it for elimination of secret-dependent branches. To that end, assignment statements in the secret-dependent control flow need to be rewritten. To preserve the constant-time properties and hide the secret-dependent memory access pattern, these assignment statements are transformed into cmov constructs. This transformation replaces the assignment operations in secret-dependent branches to avoid MOV operations. Unlike MOV operations in secret-dependent branches, cmov will

Algorithm 4 Transformation rule to rewrite early termination in a loop

Before

1: **function** F($x : T$): T_{ret}
2:     **while** $expr_{cond}$ **do**
3:         $Stmt_1$
4:         **if** $expr_{secret}$ **then**
5:             **return** $expr_{ret}$ ▷ Early-terminating secret-dependent branch
6:         **end if**
7:         $Stmt_2$
8:     **end while**
9:     **return** $expr_{retFin}$
10: **end function**

After

1: **function** F($x : T$): T_{ret}
2:     T_{ret} $return_val \leftarrow expr_{retFin}$
3:     $SECRET$ $bool$ $not_return \leftarrow true$
4:     **while** $expr_{cond}$ **do**
5:         **if** not_return **then**
6:             $Stmt_1$
7:         **end if**
8:         **if** $expr_{secret} \wedge not_return$ **then**
9:             $return_val \leftarrow expr_{ret}$
10:             $not_return \leftarrow false$
11:         **end if**
12:         **if** not_return **then**
13:             $Stmt_2$
14:         **end if**
15:     **end while**
16:     **return** $return_val$
17: **end function**

always execute and write the variable with either the old or the new value contributing to both the constant-time property and identical memory access pattern.

Algorithm 5 Matcher used to find the assignment statements in secret-dependent branches

$$\forall expr_{LHS} := expr_{RHS} : ifStmt(expr_{secret}) \wedge$$
$$(expr_{LHS} := expr_{RHS} \in Stmt_{if}$$
$$\vee\ expr_{LHS} := expr_{RHS} \in Stmt_{else})$$

The matcher in Algorithm 5 is utilized to identify all the assignment operations in the secret-dependent control flow. For our example, this matcher will match the assignment operations in lines 7-8 and 13-14 in Listing 3.1. These statements are rewritten using the transformation rule in Algorithm 6. The first argument passed to cmov() is the predicate that is the condition of the secret-dependent if statement. For the nested if statements, these conditions are accumulated by combining nested conditional expressions with conjunctions

```
1 bool pwcheck(PUBLIC std::vector<byte_t> &guess,
2                 SECRET std::vector<byte_t> &pw) {
3   bool return_val = true;
4   SECRET bool not_return0 = true;
5   SECRET bool not_return1 = true;
6   if ((guess.size() != pw.size()) & not_return0) {
7     return_val = false;
8     not_return0 = false; }
9   if (not_return0) {
10    for (size_t i = 0; i < guess.size(); i++) {
11      if (not_return1) {}
12      if ((guess[i] != pw[i]) & not_return1) {
13        return_val = false;
14        not_return1 = false; }
15      if (not_return1) {}}}
16  return return_val; }
```

Listing 3.1: After automatic elimination of early termination

(e.g., $expr_{secret} \wedge expr_{secretNested}$). The second argument is the old value, i.e., the $expr_{LHS}$, which is analogous to the branch not taken. Finally, the third argument is the new value $expr_{RHS}$ which is assigned when the branch is taken. These transformations are shown in lines 7-8 and 13-14 in Listing 3.2 shown in color blue ■.

Algorithm 6 Transformation rule to rewrite assignments in secret-dependent branches

Before

1: **function** F($x : T$)
2:     **if** $expr_{secret}$ **then**
3:         **if** $expr_{secretNested}$ **then**
4:             $expr_{LHS1} \leftarrow expr_{RHS1}$
5:         **end if**
6:         $expr_{LHS2} \leftarrow expr_{RHS2}$
7:     **end if**
8: **end function**

After

1: **function** F($x : T$)
2:     **if** $expr_{secret}$ **then**
3:         **if** $expr_{secretNested}$ **then**
4:             $expr_{LHS1} \leftarrow cmov(expr_{secret} \wedge expr_{secretNested},$
5:                         $expr_{LHS1}, expr_{RHS1})$
6:         **end if**
7:         $expr_{LHS2} \leftarrow cmov(expr_{secret}, expr_{LHS2}, expr_{RHS2})$
8:     **end if**
9: **end function**

```
1 bool pwcheck(PUBLIC std::vector<byte_t> &guess,
2               SECRET std::vector<byte_t> &pw) {
3   bool return_val = true;
4   SECRET bool not_return0 = true;
5   SECRET bool not_return1 = true;
6   if ((guess.size() != pw.size()) & not_return0) {
7     return_val = cmov((guess.size() != pw.size()) & not_return0, return_val, false);
8     not_return0 = cmov((guess.size() != pw.size()) & not_return0, not_return0,
      false);
9   if (not_return0) {
10    for (size_t i = 0; i < guess.size(); i++) {
11      if (not_return1) {}
12      if ((guess[i] != pw[i]) & not_return1) {
13        return_val = cmov((guess[i] != pw[i]) & not_return1 & not_return0,
      return_val, false);
14        not_return1 = cmov((guess[i] != pw[i]) & not_return1 & not_return0,
      not_return1, false);}
15      if (not_return1) {}}}
16  return return_val;}
```

Listing 3.2: After rewriting secret-dependent assignments using the cmov construct

3.1.3 Final Branchless Transformation

In the final step of these transformations, all of the secret-dependent branches that use the cmov construct are eliminated. Utilization of the matcher below in Algorithm 7 will match the if statements in lines 6, 9, 11, 12, and 15 in Listing 3.2. This matcher matches all the branches with cmov statements in them. Since the cmov construct already does the assignment operation depending on the predicate that is accumulated by combining the branch conditions, there is no need to keep these secret-dependent branches to protect the program's behavior. These matched secret-dependent branches are eliminated through the transformation rule shown in Algorithm 8. This transformation removes the AST nodes from the program. The final output after all the transformations is shown below in Listing 3.3.

Algorithm 7 Matcher used to find cmov calls in secret-dependent branches

$$ifStmt(expr_{secret}) \wedge (cmov(...) \in Stmt_{if} \vee cmov(...) \in Stmt_{else})$$

Algorithm 8 Transformation rule to eliminate secret-dependent branches

Before

1: **function** $\mathrm{F}(x : T)$
2: **if** $expr_{secret}$ **then**
3: **if** $expr_{secretNested}$ **then**
4: $expr_{LHS1} \leftarrow cmov(expr_{secret} \wedge expr_{secretNested},$
5: $expr_{LHS1}, expr_{RHS1})$
6: **end if**
7: $expr_{LHS2} \leftarrow cmov(expr_{secret}, expr_{LHS2}, expr_{RHS2})$
8: **end if**
9: **end function**

After

1: **function** $\mathrm{F}(x : T)$
2: $expr_{LHS1} \leftarrow cmov(expr_{secret} \wedge expr_{secretNested},$
3: $expr_{LHS1}, expr_{RHS1})$
4: $expr_{LHS2} \leftarrow cmov(expr_{secret}, expr_{LHS2}, expr_{RHS2})$
5: **end function**

```
1 bool pwcheck(PUBLIC std::vector<byte_t> &guess,
2               SECRET std::vector<byte_t> &pw) {
3   bool return_val = true;
4   SECRET bool not_return0 = true;
5   SECRET bool not_return1 = true;
6   {return_val =
7         cmov((guess.size() != pw.size()) & not_return0, return_val,
     false);
8    not_return0 =
9         cmov((guess.size() != pw.size()) & not_return0, not_return0,
     false);}
10  {for (size_t i = 0; i < guess.size(); i++) {
11      {return_val = cmov((guess[i] != pw[i]) & not_return1 &
     not_return0, return_val, false);
12        not_return1 = cmov((guess[i] != pw[i]) & not_return1 &
     not_return0, not_return1, false);}}}
13  return return_val;}
```

Listing 3.3: After eliminating secret-dependent branches

Chapter 4

Evaluation

To evaluate SOTERIA we run it on benchmarks converted from ones used by state-of-the-art side-channel analysis techniques and answer the following research questions:

RQ1. Do our benchmarks converted from Java to C/C++ still exhibit unsafe and safe behavior?

RQ2. To what extent do SOTERIA transformations result in intermediate-representation code that is constant-time on secret-dependent input?

RQ3. To what extent do SOTERIA transformations maintain the functional correctness of the original code?

We elaborate on the experiments that we conduct to answer these RQs and the results of those experiments in the remainder of this section.

4.1 RQ1: Correctness of Converted Side-Channel Benchmarks

To assess the effectiveness of Soteria in latter RQs, we need evaluation subjects that exhibit behaviors that are both unsafe (i.e., they have timing side channels) and safe (i.e., lack timing side channels). To that end, we select benchmarks utilized in the evaluation of the state-of-the-art side-channel analysis approaches DifFuzz [38], Blazer [7], and Themis [20]. However, these benchmarks are written in Java. Unfortunately, Java is a lower-value target since high-value target programs, such as cryptographic libraries, are written in C/C++ and Java's JIT compilation might introduce timing side channels. As a result, we need to convert these Java benchmarks to C/C++. To ensure accuracy of our converted benchmarks, we diligently rewrote them into C/C++ to be compatible with the clang implementation of Soteria. To test (1) the functional correctness of our converted benchmarks and (2) whether timing properties are preserved for them, a modified version of DifFuzz [38], a differential fuzzing tool to automatically detect timing side channels, is utilized. DifFuzz outperforms the other side-channel detection tools [38]. Its original implementation targets Java programs. However, DifFuzz utilizes Kelinci [29], which is based on American Fuzzy Loop (AFL) [47], a widely-used fuzzing tool that targets C/C++ programs. The generality of the DifFuzz approach and the fact that AFL already targets C/C++ enabled us to easily port DifFuzz to target C/C++ programs for our use cases. For these reasons, we decided to utilize DifFuzz to verify the impact of our transformations.

Benchmarks	# Passed Tests	# Total Tests	Percentage
Array	14873	14873	%100
LoopAndBranch	2443	2443	%100
Sanity	2073	2073	%100
Straightline	5005	5005	%100
passwordEq	15133	15133	%100
PWCheck	16371	16371	%100

Table 4.1: Correctness Results for Rewritten Benchmarks

For correctness, we compare the results of these rewritten benchmarks to the results of

original benchmarks that are written in Java using the inputs generated by the fuzzer; the result of this effort is shown in Table 4.1. The modified fuzzer that targets C/C++ programs is used to verify that rewritten benchmarks still exhibit the original unsafe and safe behavior. The original purportedly "safe" versions of two benchmarks entitled (**Array-Safe** and **LoopAndBranch**) do not actually show safe behavior. In the following section (Section 4.2), we describe these two benchmarks in more detail. These benchmarks are manually rewritten and the related results are shown in Table 4.2 with the **-Updated** suffix. We elaborate on the updates we apply to eliminate safety and functional correctness errors in the remainder of our discussion of RQ1 in this section.

4.2 Updates to Fix Errors in Original Benchmarks

As shown in Table 4.2, most benchmarks have comparable results to DifFuzz. While most benchmarks show similar timing behavior (as measured using instructions retired) there is one that does not: **Array Safe-Original**, which is shown in Listing 4.1. The `array_unsafe` function in Listing 2.1 is originally made safe by adding a dummy while loop in the false branch. However, the discrepancy in the `array_safe` function arises from the extra division operator in the false branch in line 11 of Listing 4.1. Whenever the variable `taint` is not negative this extra division operation will contribute additional instructions to the false branch per iteration creating a variable imbalance between two branches. Even this small example shows that manually writing constant-time code is not straightforward. This side-channel vulnerability is mitigated by taking the division operation out of the loop and adding more dummy statements shown in green in Listing 4.2 in both branches to balance the number of instructions resulting in a constant-time `array_safe` function, the results of which are shown as **Array Safe-Updated** in Table 4.2.

```
1 int array_safe(PUBLIC std::vector<int> &a, SECRET int taint) {
2   int t = 0;
3   if (taint < 0) {
4     int i = a.size()-1;
5     while (i >= 0) {
6       t = a[i];
7       i--;}
8   } else {
9     int i = a.size()-1;
10    while (i >= 0) {
11        t = a[i] / 2;
12        i--; }}
13  return t;}
```

Listing 4.1: Nonconstant-time array_safe

```
1 int array_safe(PUBLIC std::vector<int> &a, SECRET int taint) {
2   int t = 0;
3   if (taint < 0) {
4     int i_dummy = 0;
5     t = a[i_dummy] / 2;
6     int i = a.size()-1;
7     while (i >= 0) {
8       t = a[i];
9       i--;}
10  } else {
11    int i = 0;
12    int i_dummy = a.size()-1;
13    while (i_dummy >= 0) {
14      t = a[i_dummy];
15      i_dummy--; }
16    t = a[i] / 2; }
17  return t;}
```

Listing 4.2: Updated constant-time array_safe

Additionally, we detected that the original safe version of the **LoopAndBranch** benchmark does not have the same behavioral semantics as its unsafe counterpart. This is caused by the operation change shown in line 7 of Listing 4.3. This change forces the execution of this program to take the branch in line 8 unless the addition operation causes an overflow that will take the else branch at line 13. We rewrite a version of this program to make it safe and have the same semantics as its unsafe counterpart. The results are shown in Table 4.2 under **LoopAndBranch Safe-Updated**.

An important point to note here is that compiler optimizations were turned off when compiling these benchmark programs as they are simple programs with dummy instructions to create intentional imbalance and balance in the number of instructions in secret-dependent

control flows in safe and unsafe versions of these benchmark programs. Compiler optimizations might optimize away both manual and automatic mitigation in respectively safe and rewritten versions of these programs in both IR and machine-code optimizations passes with target-specific optimizations.

```
 1 int loopandbranch_safe(PUBLIC int a, SECRET int taint) {
 2   int i = a;
 3   if (taint < 0) {
 4     while (i > 0) {
 5       i--;}
 6   }else {
 7     taint = taint - + 10;
 8     if (taint >= 10) {
 9       i = i + taint;
10       int j = a;
11       while (j > 0) {
12         j--;}
13     }else {
14       if (a < 0) {
15         int k = a;
16         while (k > 0)
17           k--;}}}
18   return taint;}
```

Listing 4.3: Nonconstant-time loopandbranch. The red operator is unsafe code removed in the original version; the green operator is the code added that attempted to make the code safe but failed to do so.

4.3 RQ2: Constant-Time Effectiveness of Transformations

This question investigates the effectiveness of Soteria's source-to-source transformations at mitigating the side-channel vulnerabilities. After transforming code to eliminate timing side channels, we test the resulting code to detect if any timing side channels remain. To make sure that our transformations result in constant-time secret-dependent paths, we utilize our modified version of DiffFuzz. This fuzzing-based approach finds inputs to the program that maximize resource consumption between two executions. These two executions are run

with two mutated secrets while keeping the public values the same to detect the variance only in secret-dependent paths. This resource consumption is measured as the number of instructions retired, where differences depending on secret-dependent paths indicate a detected timing side channel.

To answer RQ2, two different techniques are used: 1) Inputs that are found by the fuzzer to cause discrepancies in execution time for unsafe programs are also tested by running the mitigated programs using these same inputs; and 2) these mitigated programs are also fuzzed independently from these inputs. The fuzzer was not able to detect any timing side-channel vulnerabilities in these hardened programs in which timing side-channel vulnerabilities are automatically mitigated as shown in Table 4.2.

To demonstrate the ability of SOTERIA's transformations to mitigate timing side channels, we discuss in more detail the impacts of those transformations on the running example (i.e., the unsafe password check in Listing 1.1). In Table 4.3, the column titled **Unsafe pwcheck** shows the number of instructions retired during the execution of unsafe password check; the column titled **Mitigated pwcheck** shows the number of instructions retired during the execution of the mitigated password check program that is written in a flat fashion. The first row shows us the vulnerability in lines 2-3 of Listing 1.1; when the size of `guess` and `pw` are not the same, the program terminates early. The attacker will be able observe from other input pairs that have longer execution times that the attacker-controlled `guess` buffer does not match the secret `pw` in size. This vulnerability could be exploited in the unsafe version via brute forcing until execution takes longer. However, the flat version takes 201 instructions whether or not the size of these two buffers are equal to each other, i.e., the number of instructions retired in the first and the second row are equal and cost is equal to 0 in both rows in Table 4.3.

The results for other inputs highlight the side-channel vulnerability in lines 5-6 of Listing 1.1. This secret-dependent return statement causes early termination of the program as soon

as the elements at the same index are not equal to each other. This results in a leakage from which the attacker can infer the number of leading elements that are equal to each other: the number of instructions retired are 70, 99, 128, and 142, respectively, when 0, 1, 2, and 3 elements match. As shown in the last row, the cost (difference in number of instructions) will potentially increase with longer buffers resulting in a more distinguishable variance for an attacker to exploit. On the other hand, the execution time of the transformed program only depends on the size of the `guess` buffer. This does not leak anything for the attacker to exploit since the size of the `guess` buffer is already known by the attacker.

4.4 RQ3: Functional Correctness of Transformed IR

To test the correctness of the transformations, we utilize the inputs found by the fuzzer during side-channel fuzzing, in addition to the manual unit tests we wrote during the implementation of the transformations. We run the original programs from benchmarks and the converted timing side-channel resistant ones using these inputs and then compare their outputs, checking to see if the outputs across benchmarks are the same. The inputs from the fuzzer are utilized to verify that our transformations kept the functional behavior of the programs intact—demonstrating our transformations to be functionally correct. As Table 4.1 demonstrates, Soteria achieves 100% correctness using all these test cases.

		Number of Instructions Retired					
		Soteria's Fuzzer			DiffFuzz		
Benchmarks	**Version**	**Average δ**	**Std. Error**	**Maximum**	**Average δ**	**Std. Error**	**Maximum**
Microbench							
Array	Safe-Original	48,245	120,586	528,050	1	0	1
	Safe-Updated	0	0	0			
	Unsafe	243,202	582,305	2,575,279	192	2.68	195
	Mitigated	0	0	0			
LoopAndBranch	Safe-Original	103,344,860	1,138,675,100	12,884,902,116	1,468,212,312	719,375,479.77	4,278,268,702
	Safe-Updated	0	0	0			
	Unsafe	796,740	1,958,983	7,188,937	4,283,404,852	4,450,278	4,294,838,782
	Mitigated	0	0	0			
Sanity	Safe	0	0	0	0	0	0
	Unsafe	17,349,113	12,035,726	44,770,137	4,213,237,198	60,857,888	4,290,510,883
	Mitigated	0	0	0			
Straightline	Safe	0	0	0	0	0	0
	Unsafe	806	0	806	8	0	8
	Mitigated	0	0	0			
STAC							
PasswordEq	Safe	0	0	0	0	0	0
	Unsafe	576,085	1,288,774	5,205,565	86	20.31	127
	Mitigated	0	0	0			
Running Example							
pwcheck	Safe	0	0	0			
	Unsafe	703,689	1,495,518	5,668,848			
	Mitigated	0	0	0			

Table 4.2: The results of differential fuzzing on converted benchmarks' constant-time behavior. Red cells indicate that they could not be compared since automatically mitigated and updated, safe versions are not available in the original Java benchmarks.

Public Input Value	Secret Values	Number of Instructions Retired			
		Unsafe pwcheck	Cost	Mitigated pwcheck	Cost
$guess = 1$	$pw = 12$	41	0	201	0
	$pw = 123$	41		201	
$guess = 1$	$pw = 1$	84	14	201	0
	$pw = 2$	70		201	
$guess = 12$	$pw = 12$	113	43	295	0
	$pw = 34$	70		295	
$guess = 123$	$pw = 123$	142	72	389	0
	$pw = 456$	70		389	
$guess = 123$	$pw = 193$	99	29	389	0
	$pw = 129$	128		389	
$guess = 123456$	$pw = 777777$	70	159	671	0
	$pw = 123456$	229		671	

Table 4.3: Results of the Running Example

Chapter 5

Threats to Validity

5.1 Construct Validity

The execution timing cost is measured by counting the number of instructions retired during the execution of the program by using *perf_events* [25] library on *Linux*. DifFuzz [38], Themis [20] and Blazer [7] also utilize a similar approach by counting bytecode instructions executed in Java programs. This approach, counting the number of instructions retired during the execution of the program, as opposed to measuring the wall-clock time of the execution, is a more robust metric under the assumption of the fact that these instructions are constant-time and not variable-time instructions. For the modern implementations of x86 architectures, all integer arithmetic instructions are constant time except for the division instruction [23]. Measuring the wall-clock time could be imprecise due to its sensitivity to other processes competing for the same resources. Since previous timing side-channel detection approaches use a similar measurement, counting the number of instructions enables us to have a comparable evaluation with them.

We note that while the results will not be directly comparable because (1) counting native

instructions as opposed to bytecode instructions will yield different results and (2) the fuzzing approach uses random mutations, so the inputs the fuzzers come up with will likely be different. Nevertheless, these results still indicate the correctness of these benchmarks. For example, when we look at the **Straightline** benchmark in Table 4.2, we see a similar result for both SOTERIA's fuzzer and DifFuzz. Both fuzzers have a fixed maximum and average number of instructions resulting in zero standard deviation (a maximum of 806 for SOTERIA's fuzzer and 8 for DifFuzz). As shown in Listing 5.1, this result occurs because both branches in the `straightline_unsafe` function have a fixed number of instructions (i.e., no loops with a variable number of iterations)—resulting in the same amount of difference in the number of instructions between two branches. To account for the randomness of the fuzzers, each version of the benchmark is fuzzed for 45 minutes and this experiment is repeated 5 times. Average results of these repeated experiments are shown in Table 4.2.

```
1 int straightline_unsafe(SECRET int a, PUBLIC int b) {
2   int x=a, y=b;
3   if( a > 0 && b > 0) {
4     x = 2;
5   } else {
6     // FOLLOWING CODE SNIPPET REPEATS 9 TIMES
7     x=1+y; y=2+x; x=3+y; y=4+x; x=5+y; y=6+x; x=7+y; y=8+x; x=9+y; y=10+x; x=1+y; y=2+x;
       x=3+y; y=4+x; x=5+y; y=6+x; x=7+y; y=8+x; x=9+y; y=10+x; x=1+y; y=2+x; x=3+y; y=4+x
      ; x=5+y; y=6+x; x=7+y; y=8+x; x=9+y; y=10+x;
8   }
9   return x+y;
10 }
```

Listing 5.1: Straightline_unsafe benchmark has a fixed number of instructions difference between the imbalanced branches.

5.2 Internal Threat to Validity

Internal threats to validity in our work arise in the context of validating the correctness of our converted benchmarks. Two major goals are important to consider in terms of internal validity in this context: First, to test that the program behaviors are kept intact and

maintain the same semantics as the original benchmarks; second, to show that the rewritten programs still exhibit unsafe and safe behavior. These benchmark programs—adapted from DiffFuzz, Themis, and Blazer are small applications—which have two versions: unsafe (with intentional timing side-channel vulnerabilities) and safe (side-channel vulnerabilities from unsafe versions are mitigated). To show that the benchmarks are correctly rewritten from Java to C/C++, not only are manual tests used but also the expected outputs from inputs that the fuzzer comes up with are used, and these outputs are compared to the outputs of the original Java versions. The correctness results from this comparison are shown in Table 4.1. Additionally, as discussed in Section 4.2, we corrected errors in the original benchmarks—which arose from unsafe code that is supposed to be safe and also has functional correctness issues—further demonstrating our diligence in ensuring their correctness.

5.3 External Threat to Validity

This section discusses the generalizability of Soteria's approach. Since the transformations are not language-specific, they are applicable to other languages. With a reasonable amount of engineering effort, it should be possible to integrate these source-to-source transformations that automatically rewrite Java programs using Soteria's approach. Additionally, these transformations can be done at the IR level instead of the source level. Since Soteria aims to harden general-purpose programs that are written by general programmers and not only side-channel experts such as cryptographers, we decided to implement these transformations at the source level as it would make the code-review process easier. Furthermore, the type of benchmark programs also fit this purpose as they are general-purpose programs with common optimization techniques such as the early termination of functions and loops.

Chapter 6

Related Work

With the increasing recognition of side-channel vulnerabilities, there has been a diverse body of research focusing on both detection and mitigation. There have been efforts to investigate implementing a domain-specific language to write constant-time programs. `qhasm` [13] is a portable assembly language with a register allocator that allows a programmer to write high-performing cryptographic code. `Vale` [15] is similarly a portable high-performance assembly code that utilizes a proven-correct taint-analysis engine. This engine verifies the abstract syntax tree generated from this assembly code to prove that the program is side-channel resistant. `Jasmin` [5], unlike the previously mentioned languages, combines high-level constructs such as variables, arrays and functions with low-level capabilities such as instruction selection and scheduling, and register allocation to enable a programmer to have a stronger control over the program. Jasmin utilizes EasyCrypt to formally verify that the executable is constant-time [10]. `FaCT` [18] is another DSL that helps write constant-time code using standard and high-level constructs. FaCT compiles such high-level code to constant-time LLVM intermediate representation (IR). These languages are not general-purpose programming languages and they are designed with cryptographic implementations in mind. Soteria, unlike these DSLs, takes an annotated C/C++ source code and transforms it into a constant-time

IR. With SOTERIA, writing timing side-channel resistant code is not reserved for only experts; instead, every programmer is able to harden their general-purpose programs against side-channel vulnerabilities.

There are also several efforts to formally verify the existing implementations of cryptographic libraries and build formally verified cryptographic implementations. Barthe et. al [9] performs a rigorous study of cache-timing attacks in virtualized environments by modeling virtual addressing, memory mappings, page tables, translation lookaside buffers (TLBs) and caches. The authors provide a formal proof that constant-time implementations are resistant against cache-timing side-channel attacks in this model of virtualization platforms and evaluate their work on implementations of PolarSSL AES, DES and RC4, SHA256 and Salsa20. Moreover, CacheFix [19] automatically verifies and synthesizes patches for cache side-channel vulnerabilities using symbolic verification techniques. Even though CacheFix generates proofs for the programs of their cache side-channel freedom, it focuses only on cache side-channel vulnerabilities and, unlike SOTERIA, does not consider non-cache related timing side-channel vulnerabilities such as early terminations and imbalanced secret-dependent branches. Additionally, Beringer et. al [11] verifies the correctness and security of OpenSSL implementation of HMAC with machine-checked proofs in `Coq` [22]; Appel [8] similarly verify the correctness of OpenSSL implementation of SHA-256 by utilizing `Coq` and `CompCert`, a verified C compiler. Furthermore, Ye et. al [46] verifies the correctness of mbedTLS implementation of HMAC-DRBG by formalizing the functional specification and using a hybrid game-based proof. Tsai et. al [44] verifies low-level mathematical constructs, often written in assembly and manually optimized, that are used as constant-time primitives to implement the underlying algebraic structures in an OpenSSH implementation of X25519. Almeida et. al [6] formally verifies the LLVM IRs of constant-time implementations. Besides formal verification of existing implementations of cryptographic libraries, Bhargavan et. al [14] implements a formally verified version of TLS 1.2. These research efforts to formally verify the correctness of cryptographic libraries and their resistance against side-channel attacks

are not generalizable as they target specific projects in isolation and suffer from scalability issues.

This large body of research work to formally verify these implementations relies on programmers to write these programs in a constant-time fashion to begin with. To the best of our knowledge, Soteria is the first approach to provide an end-to-end mitigation and verification of the impacts of this mitigation.

Chapter 7

Conclusions and Future Work

We have presented SOTERIA, a compiler-based approach to transform sensitive programs that are written in a non-constant-time manner into their constant-time equivalents. We believe that a sensitive system should be timing side-channel resistant overall rather than having only the high-value target components, such as cryptographic libraries, hardened against timing side-channel vulnerabilities. When programming, we generally rely on and trust compilers to generate correct and optimized programs; we should also rely on compilers to generate secure programs to protect against timing side-channel vulnerabilities. SOTERIA aims to be a step toward this goal. It is integrated into a mainstream compiler to allow a programmer to annotate the secret parts of the code and perform source-to-source transformations to automatically eliminate timing side channels. SOTERIA's results on six evaluation benchmarks that we converted from ones used by three state-of-the-art side-channel analysis approaches—DifFuzz [38], Blazer [7], and Themis [20]—demonstrate the ability of SOTERIA to mitigate timing side channels while maintaining functional correctness.

Nevertheless, mitigating these side channels at the source level is not sufficient to guarantee the constant-time properties of the program. The compiler might introduce optimizations

that revert these mitigations. This could be avoided by detecting these specific optimizations and disabling them. A sound and complete formal verification of LLVM IR of mitigated source code would give stronger guarantees for constant-time behavior [6]. We aim to adopt such an approach for future work. Besides mitigation at the IR level, mitigating timing side channels requires the knowledge of computer architecture. Constant-time implementations need to consider architecture-specific performance optimizations that might induce side-channel vulnerabilities. To address this challenge, our future work includes automatic detection and mitigation of vulnerabilities that might arise when the compiler back-end generates nonconstant-time executables from constant-time IR.

Bibliography

[1] Y. A Younis, K. Kifayat, Q. Shi, and B. Askwith. A new prime and probe cache side-channel attack for cloud computing. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, Washington, DC, USA, 10 2015. IEEE.

[2] O. Acıiçmez, W. Schindler, and c. K. Koç. Cache based remote timing attack on the aes. In *Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'07, page 271–286, Berlin, Heidelberg, 2007. Springer-Verlag.

[3] O. Aciicmez and W. Schindler. A major vulnerability in rsa implementations due to microarchitectural analysis threat, 2007. onur.aciicmez@gmail.com 13751 received 26 Aug 2007, last revised 26 Aug 2007.

[4] M. R. Albrecht and K. G. Paterson. Lucky microseconds: A timing attack on amazon's s2n implementation of tls. In M. Fischlin and J.-S. Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 622–643, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[5] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P. Strub. Jasmin: High-assurance and high-speed cryptography. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1807–1823, New York, NY, USA, 2017. ACM.

[6] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 53–70, USA, 2016. USENIX Association.

[7] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei. Decomposition instead of self-composition for proving the absence of timing channels. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 362–375, New York, NY, USA, 2017. Association for Computing Machinery.

[8] A. W. Appel. Verification of a cryptographic primitive: Sha-256. *ACM Trans. Program. Lang. Syst.*, 37(2), Apr. 2015.

[9] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 1267–1279, New York, NY, USA, 2014. Association for Computing Machinery.

[10] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub. Easycrypt: A tutorial. In A. Aldini, J. López, and F. Martinelli, editors, *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166, New York, NY, USA, 2013. Springer.

[11] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel. Verified correctness and security of openssl HMAC. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 207–221, Washington, D.C., Aug. 2015. USENIX Association.

[12] D. J. Bernstein. Cache-timing attacks on AES. `https://cr.yp.to/antiforgery/cachetiming-20050414.pdf`. [Online; accessed 2021-02-08].

[13] D. J. Bernstein. Writing high-speed software. `https://cr.yp.to/qhasm.html`, 2005. [Online; accessed 2021-02-08].

[14] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing tls with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy*, pages 445–459, 2013.

[15] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson. Vale: Verifying high-performance cryptographic assembly code. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 917–934, Vancouver, BC, Aug. 2017. USENIX Association.

[16] T. Brennan, N. Rosner, and T. Bultan. JIT leaks: Inducing timing side channels through just-in-time compilation. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1207–1222. IEEE, 2020.

[17] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, page 1, USA, 2003. USENIX Association.

[18] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan. Fact: A dsl for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 174–189, New York, NY, USA, 2019. Association for Computing Machinery.

[19] S. Chattopadhyay and A. Roychoudhury. Symbolic verification of cache side-channel freedom. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2812–2823, 2018.

[20] J. Chen, Y. Feng, and I. Dillig. Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 875–890, New York, NY, USA, 2017. Association for Computing Machinery.

[21] F. Cloutier. CMOVcc — Conditional Move Reference. `https://www.felixcloutier.com/x86/cmovcc`, 2022. [Online; accessed 2021-02-08].

[22] T. C. development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[23] A. Fog. *The microarchitecture of Intel, AMD and VIA CPUs An optimization guide for assembly programmers and compiler makers.*

[24] C. P. García, B. B. Brumley, and Y. Yarom. Make sure dsa signing exponentiations really are constant-time. Cryptology ePrint Archive, Report 2016/594, 2016. `https://eprint.iacr.org/2016/594`.

[25] B. Gregg. *perf: Linux profiling with performance counters.*

[26] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[27] S. He, M. Emmi, and G. Ciocarlie. ct-fuzz: Fuzzing for timing leaks, 2019.

[28] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel. A high-resolution side-channel attack on last-level cache. In *Proceedings of the 53rd Annual Design Automation Conference*, page 72. ACM, 2016.

[29] R. Kersten, K. Luckow, and C. S. Păsăreanu. Poster: Afl-based fuzzing for java with kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2511–2513, New York, NY, USA, 2017. Association for Computing Machinery.

[30] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution, 2018.

[31] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In N. Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[32] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, page 388–397, Berlin, Heidelberg, 1999. Springer-Verlag.

[33] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, Austin, TX, Aug. 2016. USENIX Association.

[34] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown, 2018.

[35] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622. IEEE, 2015.

[36] LLVM. Clang LibAST Matchers Reference. `https://clang.llvm.org/docs/LibASTMatchersReference.html`. [Online; accessed 2021-02-08].

[37] LLVM. Clang: a C language family frontend for LLVM. `https://clang.llvm.org/`, 2022.

[38] S. Nilizadeh, Y. Noller, and C. S. Pasareanu. Diffuzz: Differential fuzzing for side-channel analysis, 2019.

[39] openssl. Memory corruption in the ASN.1 encoder (CVE-2016-2108). `https://www.openssl.org/news/secadv/20160503.txt`. [Online; accessed 2021-02-08].

[40] openssl. Security of CBC Ciphersuites in SSL/TLS. `https://www.openssl.org/~bodo/tls-cbc.txt`. [Online; accessed 2021-02-08].

[41] openssl. Timing vulnerability in DSA signature generation (CVE-2018-0734). `https://www.openssl.org/news/secadv/20181030.txt`. [Online; accessed 2021-02-08].

[42] openssl. Constant-time Constructs in OpenSSL. `https://github.com/openssl/openssl/blob/master/include/internal/constant_time.h`, 2022. [Online; accessed 2021-02-08].

[43] J.-J. Quisquater and D. Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *Proceedings of the International Conference on Research in Smart Cards: Smart Card Programming and Security*, E-SMART '01, page 200–210, Berlin, Heidelberg, 2001. Springer-Verlag.

[44] M.-H. Tsai, B.-Y. Wang, and B.-Y. Yang. Certified verification of algebraic properties on low-level mathematical constructs in cryptographic programs. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1973–1987, New York, NY, USA, 2017. Association for Computing Machinery.

[45] 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)*, pages 719–732, San Diego, CA, 2014. USENIX Association.

[46] K. Q. Ye, M. Green, N. Sanguansin, L. Beringer, A. Petcher, and A. W. Appel. Verified correctness and security of mbedtls HMAC-DRBG. *CoRR*, abs/1708.08542, 2017.

[47] M. Zalewski. *American fuzzy lop (afl)*.