Timing Side-Channel Mitigation via Automated Program Repair

HAIFENG RUAN, National University of Singapore, Singapore, Singapore
YANNIC NOLLER, Ruhr University Bochum, Bochum, Germany
SAEID TIZPAZ-NIARI, University of Texas at El Paso, El Paso, TX, USA
SUDIPTA CHATTOPADHYAY, Singapore University of Technology and Design, Singapore, Singapore
ABHIK ROYCHOUDHURY, National University of Singapore, Singapore, Singapore

Side-channel vulnerability detection has gained prominence recently due to Spectre and Meltdown attacks. Techniques for side-channel detection range from fuzz testing to program analysis and program composition. Existing side-channel mitigation techniques repair the vulnerability at the IR/binary level or use runtime monitoring solutions. In both cases, the source code itself is not modified, can evolve while keeping the vulnerability, and the developer would get no feedback on how to develop secure applications in the first place. Thus, these solutions do not help the developer understand the side-channel risks in her code and do not provide guidance to avoid code patterns with side-channel risks. In this article, we present Pendulum, the first approach for automatically locating and repairing side-channel vulnerabilities in the source code, specifically for timing side channels. Our approach uses a quantitative estimation of found vulnerabilities to guide the fix localization, which goes hand-in-hand with a pattern-guided repair. Our evaluation shows that Pendulum can repair a large number of side-channel vulnerabilities in real-world applications. Overall, our approach integrates vulnerability detection, quantization, localization, and repair into one unified process. This also enhances the possibility of our side-channel mitigation approach being adopted into programming environments.

CCS Concepts: • **Security and privacy** → **Software and application security**; • **Theory of computation** → *Program analysis*; • **Software and its engineering** → *Software testing and debugging*;

Additional Key Words and Phrases: side-channel vulnerability, program repair, software engineering

ACM Reference format:
Haifeng Ruan, Yannic Noller, Saeid Tizpaz-Niari, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2024. Timing Side-Channel Mitigation via Automated Program Repair. *ACM Trans. Softw. Eng. Methodol.* 33, 8, Article 206 (November 2024), 27 pages.
https://doi.org/10.1145/3678169

This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair," MOE-MOET32021-0001. S. Tizpaz-Niari was partially supported by the NSF under grant CNS-2230060.
Authors' Contact Information: Haifeng Ruan, National University of Singapore, Singapore; e-mail: hruan@comp.nus.edu.sg; Yannic Noller (corresponding author), Ruhr University Bochum, Bochum, Germany; e-mail: yannic.noller@acm.org; Saeid Tizpaz-Niari, University of Texas at El Paso, El Paso, TX, USA; e-mail: saeid@utep.edu; Sudipta Chattopadhyay, Singapore University of Technology and Design, Singapore, Singapore; e-mail: sudipta_chattopadhyay@sutd.edu.sg; Abhik Roychoudhury, National University of Singapore, Singapore, Singapore; e-mail: abhik@comp.nus.edu.sg
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1557-7392/2024/11-ART206
https://doi.org/10.1145/3678169

1 Introduction

Programs often handle confidential data such as passwords, social security numbers, and medical records. While programmers take precautions such as encryption against data leaks, an attacker may still compromise the secret via a **side-channel (SC)**, i.e., observations about a non-functional property such as execution time and memory consumption. Studies show that the SC attack is practical [15, 19, 32] and imposes significant security threats. A well-known recent SC attack is Spectre [31], which exploits the speculative execution feature of microprocessors to infer confidential information. SC attacks have also been reported against the RSA algorithm [15] and Google's Keyczar library [36].

Detecting SC vulnerabilities is a challenging problem. As opposed to a functionality property, e.g., a linear-time temporal logic property, which is often interpreted over a single execution trace, the presence of SCs is a hyper-property, whose reasoning involves multiple traces. For detecting SCs, both static-analysis-based techniques [5, 18] and dynamic testing techniques [43] have been proposed. They can prove the absence of vulnerability or find a specific vulnerability. However, they do not reveal the severity of the vulnerability. This information can be derived via quantification [45, 46], which computes the potential amount of information leakage in bits, e.g., via counting the possible different SC observations (aka SC partitions) and using the min-entropy notion of quantitative information flow [54].

Once such SC vulnerabilities are detected and quantified, the next step is to develop a repair. The current automated SC mitigation techniques mainly focus on the runtime protection of software. They either use some monitoring solution to mask the vulnerability (e.g., see [30, 55]) or fix the vulnerabilities on IR or in the binary levels (e.g., see [58, 59, 61]). Neither helps the developer understand the SC risks in their software. Existing works have proposed different strategies to address SC vulnerabilities. In some domains like compression algorithms, a common approach is to simply disable compression when it is appropriate [47]. Another strategy is to entirely eliminate the SC vulnerability by removing all secret-dependent control locations [37, 61]. Such strategies are often limited to a specific application domain and significantly degrade performance.

In this work, we focus on fixing *timing* SCs, i.e., vulnerabilities that are exposed via observing execution times. We propose our approach Pendulum,[1] which uses a quantitative estimation of SC vulnerabilities to drive a fix localization. It then uses a pattern-based program repair technique to mitigate the identified vulnerability in the source code and ultimately reduces the amount of information leaks in terms of SC partitions. In our approach, the detection, localization, and repair of SC vulnerabilities are integrated as one joint process to automate an efficient development of secure software. It allows for user involvement (e.g., prioritizing fix locations that likely eliminate specific partitions or guiding the repair by providing loop bounds). At the same time, it can also operate in a fully automated fashion and present the final fix suggestions to the users.

In our evaluation, we apply Pendulum to existing subjects of SC vulnerabilities [43, 45] as well as new subjects. We show that Pendulum can mitigate the information leakage for 33 of 42 vulnerable programs, while for 26, it can entirely eliminate them. Moreover, only 5 of these 33 mitigating repairs introduce functional side effects. The repairs lead to a performance slowdown of 43.0% (average) or 2.6% (median).

In summary, our work makes the following contributions:

—the *mitigation* of *timing SC vulnerabilities* in the source code, *driven* by vulnerability *quantification* results, and hence, the *combination* of SC vulnerability detection and repair in a joint framework,

[1] After being repaired with our approach, a program that had a timing vulnerability can execute in constant time, regardless of the input, which is analogous to a pendulum's isochronism.

—focused vulnerability repairs in the *source code* (instead of any monitoring or **intermediate representation (IR)**-level solutions), which helps the developer to *understand* SC risks arising from code patterns, and
—the implementation of PENDULUM and its evaluation in terms of vulnerability mitigation, performance degradation, and potential side-effect introduction.

2 Overview

2.1 Problem Statement

Threat Model. We adapt our threat model from a chosen-message attack [33] where an adversary picks an ideal public input to compromise secret inputs in one trial. The adversary, who has access to the source code, can sample secret and public inputs on their local machine arbitrarily many times and construct an ideal public input that partitions the secret into many classes of timing observations. In the online mode, the attacker queries the target application with the best guess, observes SCs, and maps the observation to a partition of secret inputs. Based on this threat model, we define the following cost model to characterize the SC observations made by an attacker:

Definition 2.1 (Cost Model). The cost model of a deterministic program $\mathcal{P}$ is a tuple (X, Y, Σ, c) where $X = \{x_1, \ldots, x_n\}$ is the set of secret inputs, $Y = \{y_1, \ldots, y_m\}$ is the set of public inputs, $\Sigma \subseteq \mathbb{R}^n$ is a finite set of secrets, and $c : \mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{N}_{>0}$ is the cost function of the program over the secret and public inputs.

Cost Function. In this work, our cost function counts the number of executed bytecode instructions, assuming each bytecode has equal weight. Since our goal is to assess, localize, and repair timing SC vulnerabilities in the *source code*, we assume that the number of executed JAVA bytecodes is a reasonable abstraction to capture variations in the execution times, similar to existing work in this domain [5, 18, 43–46]. It is possible to have more refined cost functions that, e.g., assign different weights to different bytecode instructions according to their actual execution time. However, we opted to reuse the existing cost model from the previous work instead of studying a new one, which we will leave for future work.

Vulnerability Quantification. Our threat model requires a quantitative analysis of SCs to characterize the number of partitions. In particular, *min entropy* [54] uses the number of partitions to quantify immediate threats from SC adversaries. Formally, let $\Sigma_{Y=p} = \langle S_1, S_2, \ldots, S_k \rangle$ be the quotient space of Σ characterized by the cost observations under the public input p such that $s, s' \in S_i \implies |c(s, p) - c(s', p)| \leq \epsilon$. Given that $Y = p^*$ is the single public input that gives the maximum partitions k, the leakage $L_{\mathcal{P}}$ (measured in bits) can be computed as $\log_2 k$ based on min entropy. A program with more than one partition is considered *vulnerable* because an attacker who knows about the partitions and the characteristics of the corresponding secrets can infer information about the secrets via SC observations. Therefore, the attacker would be able to infer the subset S_i $(1 \leq i \leq k)$ of secrets to which the actual secret input s belongs by executing the program $\mathcal{P}$ with the public input p and observing the cost $c(s, p)$. If there were only one partition, there would be no possibility to distinguish secrets via this SC.

Definition 2.2 (Problem Statement). Given a vulnerable program $\mathcal{P}$ with disjoint public and secret input variables, our goal is (1) to search for a set of secret values $\{s_1, \ldots, s_q\}$ and a single public input p that characterizes the maximum partitions k (with $k \leq q$) in terms of SC observations. Then, (2) to identify a set of program locations $\Psi : \{\phi_1, \ldots, \phi_l\}$ that are the root causes of SCs. Finally, (3) to transform the program $\mathcal{P}$ at the locations Ψ into a semantically equivalent program $\mathcal{P}'$ such that the search of maximum SC partitions of $\mathcal{P}'$ leads to k' with $k' < k$, i.e., a reduced number of partitions, and hence, to a mitigation of the SC vulnerability.

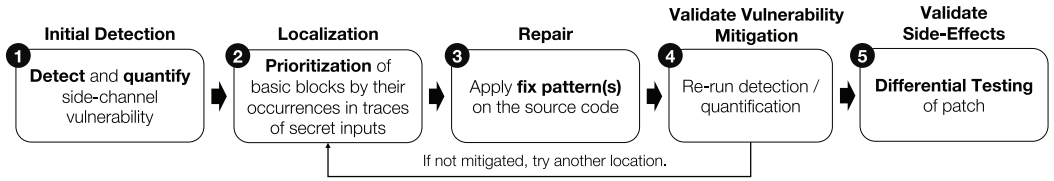

Fig. 1. Overall workflow of Pendulum.

2.2 Pendulum Overview

Input and Output. As *input*, Pendulum takes the vulnerable program and a fuzz driver that parses the input and feeds it to the program. As in previous works [43, 45], the driver is provided by a human who can distinguish public and secret inputs. The driver does not need to call the (possibly unknown) vulnerable method directly. Its implementation can follow a given template [43, 45] and be aided by driver generation tools [29, 63], which can identify entry points to vulnerable program behavior and synthesize their parameters. As *output*, Pendulum produces a repaired program with mitigated vulnerability.

Step 1—Vulnerability Quantification. As shown in Figure 1, Pendulum first detects and quantifies SCs, for which it uses QFuzz [45]. Inspired by DifFuzz [43] and AFL [62], QFuzz fuzzes the program to maximize the number of observed partitions. It randomly generates a public value and a set of secret values by mutating initial seed values. QFuzz then executes the program with the public value and each secret value, observing the corresponding cost. If the values result in more partitions than all previously generated values, QFuzz would keep them as seeds for future mutation. As output, QFuzz provides a public value and a set of secret values that lead to the most partitions. Pendulum retrieves the set of secret values that expose the most partitions and uses them for fix localization.

Step 2—Fix Localization. Pendulum compares the execution traces of the secrets pairwise to identify vulnerable basic blocks, which are then mapped to source locations. A subset of the locations are selected for repair.

Step 3—Repair. At every selected fix location, according to the program construct that causes the vulnerability, Pendulum transforms the program with a particular fix pattern. For example, the `&&` operator is substituted with a wrapper of `&`.

Step 4, 5—Validation. The repair is validated from two aspects. First, QFuzz is re-run to check if the repaired program shows reduced SC partitions (step 4). If this fuzz campaign shows that there are still too many partitions, the fuzzer input from this campaign can be used to drive yet another round of localization and repair (steps 2 and 3). This validation–repair loop can iterate until the vulnerability is reduced to an acceptable level. Secondly, the repair is checked for (the absence of) any change in functionality (step 5). In our experiments, we manually examined every repair and also performed regression testing using EvoSuite [24] and formal verification using SymDiff [35].

Envisioned Usage. We envision that Pendulum will help developers discover, understand, and mitigate timing SC vulnerabilities, by localizing potential vulnerabilities and proposing the corresponding patches. While the patch generation can be fully automated, aiming to fix all detected issues, the developer can also select repair locations from the vulnerability locations identified by Pendulum. This allows the developer to balance security improvements and performance degradations, which usually represents a trade-off in security patching. Once Pendulum generates a patch,

```
1  modPow(base, exponent, modulus, width) {
2    BigInteger r0, r1 = BigInteger.ONE, base;
3    for (int i = 0; i < width; i++) {
4      if (!exponent.testBit(width - i - 1)) {
5        r1 = fastMultiply(r0, r1).mod(modulus);
6        r0 = r0.multiply(r0).mod(modulus);
7      } else {
8        r0 = fastMultiply(r0, r1).mod(modulus);
9        r1 = r1.multiply(r1).mod(modulus);
10     }
11   }
12   return r0;
13 }
14 fastMultiply(x, y) {
15   ...
16   if (y.equals(BigInteger.ONE)) {
17     return x;
18   }
19   BigInteger z = ...;
20   return z;
21 }
```

Listing 1. Unsafe Exponentiation (Vulnerabilities Marked in Red).

the developer can inspect, test, and modify it to ensure the program semantics remain unchanged before finally applying the patch.

2.3 Illustrative Example

We explain how Pendulum works on a method computing modular exponentiation, which is common in public-key cryptography and is critical to security. The method originates from one of our experimental subjects, blazer_modpow2 [5]. In Listing 1, the method `modPow` computes the remainder when `base` (public) is raised to the power of `exponent` (secret) and divided by `modulus` (public). Two intermediate values, `r0` and `r1`, are updated iteratively for `width` times, where `width` is the bit length of `exponent` and also public.

Suppose a QFuzz run gives the public input `base=8`, `modulus=35`, `width=4`, and two different secret exponents, `exp1=8` (`1000` in bits) and `exp2=12` (`1100` in bits), which expose two different SC partitions. Pendulum would compare the execution traces of the two secrets; the locations where they diverge are considered the fix locations. There are two divergences in this case. One occurs in the second iteration, in line 4, when `exp1` and `exp2` test differently for their second highest bits. The two traces do not rejoin until the third iteration, when the third highest bits of both exponents test false. Thus they enter the true branch in line 5 and call `fastMultiply(r0,r1)`. At the point, `exp1` has `r0=29,r1=22` and takes the false branch in line 16, while `exp2` has `r0=22,r1=1` and takes the true branch. Therefore, line 16 sees another divergence. Afterward, the two traces rejoin at the fourth iteration and diverge no more.

To mitigate the vulnerability, Pendulum applies fix patterns at the two locations, as in Listing 2. At the first fix location, starting from line 4, the original `if` statement is replaced with a series of conditional assignments to `r0` and `r1`, where we have introduced two utility methods, `not` and `ite`, which perform logical negation and if-then-else respectively. The two utility methods are constructed in such a way that they execute in *constant* time, regardless of the arguments. At the second fix location, starting from line 15, the originally early-returned value `x` is conditionally stored in a new variable `result`, and the return is postponed to the end of the method. For this

```
1  modPow(base, exponent, modulus, width) {
2    BigInteger r0, r1 = BigInteger.ONE, base;
3    for (int i = 0; i < width; i++) {
4      boolean b = not(exponent.testBit(width - i - 1));
5      r1 = ite(b, fastMultiply(r0, r1).mod(modulus), r1);
6      r0 = ite(b, r0.multiply(r0).mod(modulus), r0);
7
8      r0 = ite(b, r0, fastMultiply(r0, r1).mod(modulus));
9      r1 = ite(b, r1, r1.multiply(r1).mod(modulus));
10   }
11   return r0;
12 }
13 fastMultiply(x, y) {
14   ...
15   BigInteger result = null;
16   boolean yIsOne = y.equals(BigInteger.ONE);
17   result = ite(yIsOne, x, result);
18   BigInteger z = ...;
19   return ite(yIsOne, result, z);
20 }
```

Listing 2. Safe Exponentiation (Repairs Marked in Green).

example, our experiment shows that the fix patterns eliminate the vulnerability while preserving the program functionality.

In the rest of this article, we describe the fix localization algorithm in Section 3 and the fix patterns in Section 4.

3 Fix Localization

Fix localization helps achieve maximal vulnerability mitigation with minimal program change. Given a set of secret values that reveal SC partitions, our fix localization works in two steps. First, execution traces of the secrets are compared pairwise to find partition-inducing basic blocks, called *leaky blocks*. Then, the leaky blocks are mapped to the source level.

3.1 Fix Localization at Bytecode Level

Algorithm 1 shows how to identify leaky blocks. For each pair of secrets s_i and s_j representing two different partitions, we first run the program $\mathcal{P}$ with the secrets to get their respective execution traces t_i and t_j, where a trace is a sequence of basic blocks. Then, t_i and t_j are compared to identify the set $D_{i,j}$ of blocks where the traces diverge. These are leaky blocks. The final set B of leaky blocks is the union of all such $D_{i,j}$.

At the core of Algorithm 1 is the trace comparison function, `diffTraces`. The two input traces, t_1 and t_2, are expected to start from the same basic block, which is the first block of the method that is used as the entry point in the fuzz driver. From there, the comparison goes along block by block (line 11–16) until the two traces diverge. The last basic block, `last`, where t_1 and t_2 coincide, is identified as leaky (line 18–19).

After they diverge at `last`, t_1 and t_2 may or may not rejoin each other. To determine this, we take the control flow graph of the method where `last` resides, insert an exit block as the successor of all blocks ending with a return instruction, and then compute the post-dominator tree of this graph. It is worth noting that, in constructing the control flow graph, we take care to end a basic block at any method invocation instruction, e.g., `invokespecial`, regardless of whether it returns.

Algorithm 1: Determining Fix Locations at Bytecode Level by Comparing Traces Pairwise

Input : Program $\mathcal{P}$, public value y, secret values $s_1, s_2, \cdots, s_k$
Output: Set B of basic blocks to fix

```
1  B ← ∅
2  for i ← 1 to k do
3      for j ← i + 1 to k do
4          t_i ← P(y, s_i), t_j ← P(y, s_j)
5          D_{i,j} ← diffTraces(t_i, t_j)
6          B ← B ∪ D_{i,j}
7  return B

8  Function diffTraces(t_1, t_2):
9      D ← ∅, i ← 0, j ← 0, stack ← empty stack
10     while true do
11         if t_1[i] = t_2[j] then
12             if t_1[i] ends with method invocation then
13                 stack.push(i)
14             else if t_1[i] ends with return instruction then
15                 stack.pop()
16             i ← i + 1, j ← j + 1
17         else
18             last ← t_1[i − 1]
19             D ← D ∪ { last }
20             next ← postDominator(last)
21             if next = EXIT then
22                 if stack.empty() then
23                     break
24                 else
25                     l ← stack.pop()
26                     callSite ← t_1[l]
27                     next ← postDominator(callSite)
28             while t_1[i] ≠ next do
29                 i ← i + 1
30             while t_2[j] ≠ next do
31                 j ← j + 1
32     return D
```

This facilitates maintaining a call stack as we follow the traces, which is of use later. Depending on the post-dominator of `last`, three cases may follow:

Case 1. In the simple scenario, the post-dominator of `last` is a "normal" block that is not the exit block (line 20). It follows that t_1 and t_2 will join again at this post-dominator designated as `next`. Therefore, we find `next` in the rest of both traces (line 28–31) and resume comparison from there.

Algorithm 2: Mapping a Leaky Block to a Source Element Chain

Input : *block*: An identified leaky block to fix
I: A list of bytecode instructions in the method of *block*
line_table : $I \to \mathbb{N}^+$: Line number table of the method that maps each bytecode instruction to source line number
AST: Abstract syntax tree of the method
Output: The source element chain corresponding to the block

```
1 branch ← the last instruction of block                    // must be a conditional branch
2 line_no ← line_table(branch)
3 instructions ← ⟨instruction ∈ I | line_table(instruction) = line_no⟩
4 statements ← ⟨node ∈ AST | node is a statement in line line_no⟩
5 chains ← ⟨⟩                                                // "< >" for empty list
6 for s in statements do
7     chains ← chains + parse_statement(s)                   // "+" for list concatenation; see
        Algorithm 3 for parsing
8 k ← index of branch in instructions
9 return the k-th item of chains
```

Case 2. The post-dominator is the exit block, i.e., t_1 and t_2 will not join again before returning from the method where `last` resides. We now check the call stack maintained along the way (line 11–15): if this method was invoked by some other method, i.e., it is not the entry method, t_1 and t_2 will join at the block immediately after the call site (line 25–27), where we resume comparison.

Case 3. The post-dominator is the exit block, as in the previous case, but the call stack is now empty (line 22), meaning that t_1 and t_2 will exit the entry method without ever joining again. Thus, we terminate the comparison (line 23).

3.2 Fix Localization at Source Level

Source-level localization involves mapping leaky blocks, or rather, their ending instructions, to the source code. Such instructions that allow more than one successor (remember that a leaky block causes execution divergence) include conditional branch instructions (e.g., `ifne`), `tableswitch`, and `lookupswitch`. As `switch` statements can be converted to `if` statements in principle, we consider only conditional branch instructions.

Program constructs involving conditional branches include

- —logical and comparison operators, including the not operator (`!`), binary comparison operators (`>`, `<`, `>=`, `<=`, `!=`, `==`), binary logical operators (`&&`, `||`), and the ternary operator (`?:`);
- —`if` statements;
- —loop statements, including `for`, `while`, and `do-while`.

We note that for one conditional branch in the bytecode, there is possibly a *chain* of responsible source code elements, instead of a single one. This has to do with how the Java compiler generates instructions. As an example, in the statement `if(!b)foo();`, `if` and `!` jointly produce an `ifne` instruction, which would persist unless both elements are gone; for example, `if(b)foo();` still has one `ifeq` instruction. In this case, we have a chain of two elements to be fixed.

We show our overall source-level fix localization algorithm in Algorithm 2. It maps a leaky block to its corresponding source element chain. If multiple blocks are identified as leaky, the mapping

algorithm can be applied to each block individually. In addition to the leaky block, the algorithm also takes as input all the bytecode instructions of the method where the block is located, as well as the line number table and **abstract syntax tree (AST)** of the method. The line number table can be found in the class file. It maps each instruction to a line number n, which means the instruction is compiled from source code in line n.

Algorithm 2 starts by analyzing the bytecode. It first retrieves the last instruction of the leaky block and looks up the line number of the instruction from the line number table (lines 1—2). Note that in the source line indicated by this number, there may be multiple chains, thus leading to multiple conditional branch instructions. To decide which chain corresponds to the instruction we focus on, we first get the list of instructions in the line in the order they appear in the class file (line 3). We then parse the program statements in that line (usually a single statement) sequentially to obtain the list of all chains in the line (lines 4—7). These chains are arranged in the same order they are compiled to bytecode. This is a property maintained by our customized parsing algorithm (Algorithm 3, discussed in the next paragraphs). Therefore, if the instruction we focus on is the kth of the list of instructions, it would correspond to the kth chain (lines 8–9 of Algorithm 2).

An important step in the algorithm is to parse a statement for the chains of potentially leaky elements. By examining the bytecode of programs that contain conditional branches, we have extracted rules about what source elements compile to conditional jump instructions, in what order, and how the elements are chained. Following the observed rules, we propose an algorithm for finding the chains in a statement, shown in Algorithm 3.

In Algorithm 3, each source element, or location, is represented by a `Loc` object (line 1). A `Loc` object has a `type`, e.g., `do-while` statement or a ternary operator. The type determines how the location should be repaired. Each chain is represented by a linked list of `Loc`. A `Loc` also has a `next` pointer that points to another `Loc`. When `next` is not `null`, it means the current element is chained with the `next` element. To illustrate, the chain in the `if(!b)foo()` example would be `Loc(NEGATION, next)`, where `next` refers to `Loc(IF, null)`.

The main body of Algorithm 3 is the `parse` function. It recursively searches for all chains under an AST `node`. It also takes a `next` parameter, which (if not `null`) represents a chain discovered in the previous search. Depending on the current node's type, the `parse` function may extend the `next` chain and create new chains. Algorithm 3 also defines the function `parse_statement` used in Algorithm 2, which simply calls `parse` with the `next` parameter being `null`.

Source elements that can become the `next` include `if` statements, loop statements, conditional and/or operators, and the ternary operator. For example, lines 5–10 in Algorithm 3 parse a `while` statement. The `while` statement is represented by `here` in line 6. In line 7, the condition `cond` of the `while` statement is parsed recursively, with `here` as the `next` parameter. If there is a conditional jump in `cond`, then that conditional jump (e.g., a negation expression, see line 12) may be chained with this `while` statement and added to the result list (line 8). On the other hand, if no conditional jump is found in `cond` (e.g., when the condition is a single Boolean variable), then the `while` statement itself would be added to the result list as a single-node chain. Finally, the statements enclosed in the `while` statement are parsed recursively (lines 9—10). Other types of elements, including the not operator (line 11) and binary comparison operators (line 21), are not passed down the AST as the `next` parameter but can be appended to an existing `next` chain or form a chain on their own.

As an example of the overall source-level fix localization process, we explain our fix localization for the `openmrs` subject taken from the benchmark of QFuzz [45], as shown in Listing 3. Our bytecode-level localization indicates a single leaky block, which ends at instruction 30, a conditional jump. According to the line number table shown in Listing 4, this instruction, as well as two other conditional jumps (instructions 41 and 52), belongs to line 65. In Figure 2, we highlight the three instructions in the control flow graph of the method. We then parse line 65 to find the source

Algorithm 3: Finding Potential Fix Location Chains in a Program Statement at Source Level

Input : statement: AST node that represents a statement
Output : List of source element chains

```
1  return parse(statement, null) Function Loc(type, next):
2      return a source location of type type that is chained with next
3  Function parse(node, next):
4      list ← ⟨⟩                                                    // "< >" for empty list
5      if node ~ while(cond) { block } then                  // similar for do/for/if statement
6          here ← Loc(WHILE, null)
7          locations ← parse(cond , here) ≠ ⟨⟩ ? parse(cond, here) : ⟨here⟩
8          list ← list + locations                                  // "+" for list concatenation
9          for statement in block do
10             list ← list + parse(statement, null)
11     else if node ~ !cond then
12         here ← Loc(NOT, next)
13         locations ← parse(cond, next) ≠ ⟨⟩ ? parse(cond, next) : ⟨here⟩
14         list ← list + locations
15     else if node ~ lhs && rhs then                                  // similar for ||
16         here ← Loc(AND, null)
17         lhs_locations ← parse(lhs, here) ≠ ⟨⟩ ? parse(lhs, here) : ⟨here⟩
18         here ← Loc(AND, next)
19         rhs_locations ← parse(rhs, here) ≠ ⟨⟩ ? parse(rhs, here) : ⟨here⟩
20         list ← list + lhs_locations + rhs_locations
21     else if node ~ lhs > rhs then                    // similar for <, >=, <=, ==, !=
22         list ← list + parse(lhs, null) + parse(rhs, null) + ⟨Loc(GREATER_THAN, next)⟩
23     else if node ~ cond ? lhs : rhs then
24         here ← Loc(TERNARY, null)
25         list ← list + parse(cond, here) ≠ ⟨⟩ ? parse(cond, here) : ⟨here⟩
26         list ← list + parse(lhs, next) ≠ ⟨⟩ ? parse(lhs, next) : ⟨next⟩
27         list ← list + parse(rhs, here) ≠ ⟨⟩ ? parse(rhs, next) : ⟨next⟩
28     else if node ~ (expr) then
29         list ← list + parse(expr, next)
30     else
31         for child in child nodes of node do
32             list ← list + parse(child, null)
33     return list
34 Function parse_statement(statement):
35     return parse(statement, null)
```

elements that compile to the three instructions. The two `||` operators would respectively lead to instructions 30 and 41, deciding which ones of the three Boolean expressions are executed. There would also be a third conditional jump (instruction 52), deciding whether `true` or `false` is the return value. This instruction results from both `||` operators because it would persist if either operator remains. Finally, since the leaky block ends with instruction 30, we can decide to fix the first `||` operator. The parsing rule of other types of elements are also listed in the `parse` function.

```
60  public static boolean hashMatches(String hashedPassword, String passwordToHash) {
61      if (hashedPassword == null || passwordToHash == null) {
62                  throw new APIException("password.cannot.be.null", (Object[]) null);
63      }
64
65      return hashedPassword.equals(encodeString(passwordToHash)) || hashedPassword.equals(
            encodeStringSHA1(passwordToHash)) || hashedPassword.equals(incorrectlyEncodeString
            (passwordToHash));
66  }
```

Listing 3. Vulnerable Method in org.openmrs.util.Security (with Original Line Number).

```
line 61: 0
line 62: 8
line 65: 22
```

Listing 4. Line Number Table of the Method in Listing 3.

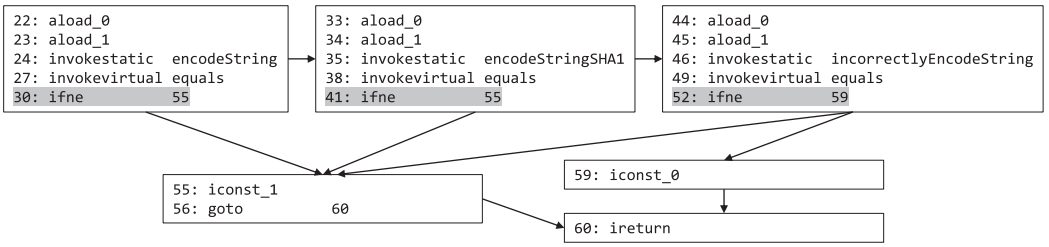

Fig. 2. Control flow graph of line 65 of the method in Listing 3 (conditional jumps highlighted in gray).

We denote this bytecode-to-source mapping with $sLoc$: for a basic block b ending in a conditional branch instruction, $sLoc(b)$ is the chain of source code elements it corresponds to. Let $L_{i,j} = \{sLoc(b) \mid b \in D_{i,j}\}$ and $L = \bigcup L_{i,j}$.

3.3 Fix Location Selection

After identifying the vulnerable locations, it remains to select one or more of them for repair. The selection can be either automated or manual. For example, the manual selection could be driven by the trade-off between vulnerability mitigation and performance degradation. In this work, we propose an automated selection procedure giving the top-n fix locations that lead to the greatest vulnerability mitigation (i.e., partition reduction).

Algorithm 4 shows the selection algorithm. The core of the algorithm is the `remainingPartitions` function, which computes the number of partitions that remain after the set F of fix locations are fixed. For two different partitions represented by secrets s_i and s_j, $L_{i,j}$ is the set of fix locations identified by comparing the execution traces of the two secrets. If these locations are repaired, s_i and s_j are expected to have the same execution time, i.e., their respective partitions are merged into one. With this, `remainingPartitions` checks, for each pair of s_i and s_j, if $L_{i,j}$ is subsumed by F. If so, s_i and s_j can be merged. After all merges, the number of remaining partitions is returned. The main body of Algorithm 4 leverages this function and searches for an n-element subset of fix locations that minimizes the number of partitions left. Note that this strategy may not lead to an optimal solution, but we found that this approach works efficiently and generates semantic preserving repairs in most cases.

Algorithm 4: Selecting Fix Locations

Precondition: $s_1, \cdots, s_k$ represent k different secret partitions
Input: $L_{i,j}$ for all $1 \le i < j \le k$ shows set of fix locations obtained by comparing the traces of s_i and s_j
n is the number of locations to fix
Output: Set F_n^* of fix locations leading to the fewest remaining partitions, where $|F_n^*| = n$

```
1  L ← ⋃ L_{i,j}
2  S ← { F ∈ 2^L | |F| = n }
3  F*_n ← argmin_{F∈S} remainingPartitions(F)
4  return F*_n

5  Function remainingPartitions(F):
6      for i ← 1 to k do
7          parent_i ← i
8      for i ← 1 to k do
9          for j ← i + 1 to k do
10             if L_{i,j} ⊆ F then
11                 parent_j ← parent_i
12     P ← { parent_i | 1 ≤ i ≤ k }
13     k' ← |P|
14     return k'
```

4 Fix Patterns

We design fix patterns to repair the vulnerable program at the selected fix locations. We attempt to retain the program logic while executing a constant number of instructions.

4.1 Fix Pattern 1: Logical and Comparison Operators

We design constant-time utilities to substitute unsafe operators, as exemplified in Listing 5. For the not, binary comparison, and ternary operators, the safe utility contains two `if` statements with opposite conditions; their functionalities depend on the argument, but their costs always offset each other. For example, line 3 in Listing 5 compiles into an `ifeq` instruction followed by a fall-through block that does the assignment. The assignment is executed if `b` is true and skipped otherwise. Line 4 compiles alike, but the assignment is executed precisely when the assignment in line 3 is skipped. Thus, the original operation is fulfilled with a constant number of executions. Note that the two `if`'s cannot be replaced with an `if-else` statement, which appears "balanced" but would have an extra `goto` instruction in the then-branch. For the logical operators `&&` and `||`, the utility methods are a wrapper for `&` and `|` respectively. For one thing, this gets rid of conditional branch instructions. For another, the second operand, which may be short-circuited [10], i.e., skipped, depending on the value of the first operand (which may in turn depend on a secret), now becomes an argument of the utility and thus is always evaluated.

Note that these repairs are for the *operator*; *operands* are another matter. Should the operands be vulnerable, they would undergo separate repair. For example, if an operand is a secret-dependent method call, the body of the method would be identified as vulnerable by the localization and repaired.

```java
public static boolean not(boolean b) {
    boolean t = false;
    if (b)  t = false;
    if (!b) t = true;
    return t;
}

public static <T> T ite(boolean cond, T t1, T t2) {
    /* substitute for ternary operator (?:) */
    T t = null;
    if (cond)  t = t1;
    if (!cond) t = t2;
    return t;
}
```

Listing 5. Constant-Time Utility Method Examples.

```
+  boolean earlyReturn = false;
+  RT returnValue = DEFAULT_VALUE;
   ...
   if (condExp) {
     ...
-    return x;
+    returnValue = x;
+    earlyReturn = true;
   }
   ...
-  return y;
+  return ite(earlyReturn, returnValue, y);
```

Listing 6. Eliminate Conditional Return.

4.2 Fix Pattern 2: IF Statement

The `if` statement is fixed in two passes. In the first pass, any `return`, `break`, or `continue` within the `if` statement is transformed into a series of new variable declarations and variable assignments. The second pass then rewrites the assignments and eliminates the `if` keyword altogether. Below, we only describe the fix pattern for early return; the patterns for `break` and `continue` are similar.

Pass 1—Fixing Early Return. As shown in Listing 6, we first declare two new variables at the beginning of the method. One is `earlyReturn`, a flag of whether an early return has occurred. The other is `returnValue`, holding the return value in case of an early return. Second, we replace the return statement with assignments that set the flag and store the early-returned value. Finally, for every other return statement, we make their return value a choice that is dependent on the `earlyReturn` flag. This is accomplished via the constant-time `ite` utility (see Listing 5). We note that this repair pattern also works if the early return is within the else-branch.

Pass 2—Fixing `if`. In the second pass, all assignment statements within both branches are made conditional via the `ite` utility, as shown in Listing 7. In the then-branch, when the guarding condition is false, an assignment is reduced to a no-op; the original assignment is executed only when the guarding condition is true. The reverse is true for assignments in the else-branch. After this, the `if` and `else` keywords can be safely removed.

```
+  boolean cond = condExp;
-  if (condExp) {
     ...
-    var1 = exp1;
+    var1 = ite(cond, exp1, var1);
     ...
-  } else {
     ...
-    var2 = exp2;
+    var2 = ite(cond, var2, exp2);
     ...
-  }
```

Listing 7. Eliminate if Statement with Conditional Move.

```
+  int ub = estimatedLoopBound;
-  for (...; condExp; ...) {
+  for (...; --ub > 0; ...) {
+    if (!condExp) {
+      break;
+    }
     ...
   }
```

Listing 8. Pass 1 of Loop Repair: Fix Number of Iterations w.r.t Upper Bound.

4.3 Fix Pattern 3: Loop Statements

A loop statement causes timing vulnerabilities when it iterates for a number of times that varies with the secret input values. As such, the repair is to fix the number of iterations to a secret-independent value. This value should be an upper bound of the number of loop iterations to preserve its functionality. It follows that, after repair, the loop may have some extra iterations compared to before. We then make these extra iterations futile by means of a conditional assignment, thus ensuring an unchanged functionality. Our fix pattern consists of two passes. As shown in Listing 8, the first pass uses a loop counter based on the upper bound `ub`. This bound can be given by the user or derived with automatic program analysis [27, 53]. Meanwhile, the original loop guard, `condExp`, is moved into the loop body to guard a newly added `break` statement. At this point, the number of iterations remains unchanged because the conditional break would always take place before `ub` reaches zero. But we have transferred the uncertainty from the loop guard to the conditional break, for which we have a fix already. Hence, in the second pass, we fix the added `if` statement as in Section 4.2. This allows the loop to execute some futile iterations to reach the estimated loop bound, making the loop safe. Note that `while` and `do-while` loops can be transformed similarly.

4.4 Discussion of Repair Technique

Reason for Pattern-Based Repair. We adopt pattern-based repair because timing SC vulnerability has relatively few types of root causes (as listed in Section 3.2), each of which can be tackled with a definite pattern. In this case, pattern-based repair is far more efficient and reliable than other approaches, e.g., search-based repair.

Side Effect. We try to design fix patterns that are free of side-effects, i.e., preserving the semantics of the program-to-fix. This is the case for a wide range of, albeit not all, real-world programs, as

shown by our evaluation in Section 5. Here, we delineate a *sufficient* condition for our fix patterns to be side-effect-free, which most of the programs in our evaluation satisfy. Because our fix patterns can change the number of times a code snippet is executed (e.g., when repairing a loop), it is desirable if the snippet either does not change the value of existing variables (so it can repeatedly execute without having an effect), or only assigns to variables explicitly (so our fix pattern can guard the assignment with a condition when needed). Such a property is captured by the concept of *purity*. Purity has varied definitions [40, 51, 52]. For our purpose, we say that an expression is pure if it does not mutate any object or variable existing before its execution. A statement is pure if it is an assignment statement with a pure expression on the right-hand side. Note, however, that a pure expression is allowed to create new objects, making this definition rather flexible. With this definition of purity, we state the following sufficient conditions for our fix patterns to be side-effect-free:

Pattern 1. the expression is pure;

Pattern 2. (a) the `if` condition is a pure expression;
(b) the branches of the `if` statement consist only of pure statements (except the ending `return`/`break`/`continue` statement, if existent);
(c) these statements do not throw an exception;
(d) if an ending `return` statement exists and returns an expression, the returned expression is pure;

Pattern 3. (a) the loop condition is a pure expression;
(b) the loop body consists only of pure statements;
(c) these statements do not throw an exception.

We have formally verified that, given these conditions, applying our fix patterns does not change program semantics. The verification is via translating the pre- and post-repair code (e.g., Listing 7) into BoogiePL [20], specifying the purity of relevant expressions and statements, and checking their equivalence with SymDiff [35]. The proof script can be found in our replication package. Note that side effects may occur when these conditions do not hold. For example, after the snippet `if(o!=null)x+=o.size();` is transformed to `cond=neq(o,null);x=ite(cond,x+o.size(),x)`, a null pointer exception may arise because the `o.size()` would be executed even if `o` is `null`. To mitigate the problem, purity analysis [40, 51, 52] can be performed to warn about the dissatisfaction of these conditions. Also, more sophisticated fix patterns can be designed, which we leave to future work.

Moreover, we emphasize that a developer can investigate the side effects of a generated patch before applying it. A fully automatic and effective way to detect side effects is to perform regression testing with test generation tools, e.g., EvoSuite [24]. The developer can also perform formal verification, e.g., with SymDiff [35], to rigorously verify that the patched program is equivalent to the original program. Finally, the developer can manually inspect the patch. In our experiment, we performed both regression testing and formal verification on the patches generated by Pendulum. The detection results agree with our manual inspection, demonstrating the feasibility of the two approaches.

Necessity of Fix Localization. As our repair is based on patterns, it may seem that our fix localization can be replaced with a plain search for anti-patterns in the source code, e.g., `if` statements. Nevertheless, such an approach would be prone to false-positives due to the great abundance of the anti-patterns, which are no more than basic program constructs. In contrast, our localization algorithm essentially performs a dynamic dependence analysis and thus gives more accurate fix locations.

5 Evaluation

We explore the following research questions.

RQ1 *(Fix localization),* Can Pendulum find the correct fix locations for the SC vulnerabilities?
RQ2 *(Vulnerability mitigation),* To what extent does Pendulum mitigate the SC vulnerabilities?
RQ3 *(Side effect),* Does Pendulum preserve the functionality of the program-to-fix?
RQ4 *(Time and space impact),* How do the generated patches influence the execution time of the programs? How large are the patches?

For RQ1–3, we also evaluate DifFuzzAR [37] as a baseline. DifFuzzAR is the state-of-the-art source-level timing SC repair tool for Java that uses the bytecode to detect the vulnerability. Therefore, it is the closest work to ours. Other related work include Debreach [47], Raccoon [49], and SC-Eliminator [61]. We do not compare with Debreach because it works only for compressor programs. We do not compare with Raccoon and SC-Eliminator, either, because they operate on LLVM IR rather than Java bytecode. To use Raccoon or SC-Eliminator for Java bytecode, one would need to translate bytecode into LLVM IR, apply the repair tool, and then translate the repaired LLVM IR back into bytecode. To the best of our knowledge, there is no tool that performs the translations reliably, thus precluding the comparison against Raccoon and SC-Eliminator. Moreover, note that we do not compare Pendulum with SC *detection* techniques like Antonopoulos et al. [5], Chen et al. [18], and Bang et al. [7] because Pendulum focuses on the localization and repair of vulnerabilities. For vulnerability detection and quantification, we choose QFuzz [45] because it represents the state of the art and is aligned with our threat model in contrast to the adaptive threat model targeted by Bang et al. [7].

Our tool, all subjects, and experimental results are available at:

https://doi.org/10.6084/m9.figshare.20731846

5.1 Implementation Details

For vulnerability detection, we use QFuzz [45]. We replace its instrumentation with Javassist [11], which computes a more accurate control flow graph. For fix localization, Javassist is used to do the instrumentation required for collecting execution traces. The map from leaky blocks to the source code is derived by analyzing the line number table of the class file and the AST of the source code using TBar [38]. Finally, the fix patterns are implemented as transformations of the AST.

5.2 Evaluation Methodology

Subject Programs. For our evaluation benchmark, we use all subjects from QFuzz [45] that have timing SC vulnerabilities; those with no or other types of vulnerabilities are excluded. We also exclude the *Leak Set* subjects, as they are not real-world programs but show their results in our artifact. Additionally, we extend the benchmark with four vulnerable programs from well-known Java security projects.

SC Types. We have manually examined the subjects for their vulnerability types (Table 1, Column *Type*). According to Section 3.2, we look for secret-dependent unsafe operators, `if` statements, and loop statements. We also indicate whether the `if` statements contain a `return`, `break`, or `continue`.

Evaluation Metrics. For RQ1, we compare the identified fix locations with that of the developer fix. For RQ2, we compare the number of SC partitions between the original program, the Pendulum-fixed program, and the developer fix. For RQ3, we first perform regression testing with EvoSuite [24] to detect changed functionality. For patches that pass this step, we additional perform a formal

Table 1. Results for the Fix Localization, the Vulnerability Mitigation, and the Side-Effect Testing

Subject	Type[a]						Fix Locations[b]		Side-Channel Partitions[c]				Regression Test Failure[e]		
	Op	If	Ret	Cont	Brk	Lop	PDL	DFZ	Orig	PDL	DFZ	Dev[d]	PDL	DFZ	Total
apache_ftpserver_clear	✓	✓	✓				1 ⊥	ID	17	1	ID	1	0	ID	17
apache_wss4j	✓	✓	✓				1 −	ID	17	1	ID	-	0	ID	10
blazer_array	✓	✓					1 =	1 =	2	1	1	2	0	0	17
blazer_login	✓	✓	✓				1 ⊂	4 *I*	17	1	1	1	0	1	7
blazer_modpow1	✓	✓					1 =	1 =	20	1	12	12	0	0	29
blazer_modpow2	✓	✓	✓				3 ⊥	1 ⊥	53	1	45	14	0	0	34
blazer_passwordEq	✓	✓	✓				1 =	2 ⊃	17	1	9	9	0	0	6
blazer_straightline	✓	✓					1 ⊥	1 ⊥	2	1	1	1	0	0	23
cryptomator_authfile	✓	✓	✓				1 =	ID	3	1	ID	1	-	-	-
Eclipse_jetty_1	✓	✓	✓				1 −	1 −	17	1	8	-	0	0	7
Eclipse_jetty_2	✓						1 −	3 −	8	1	8	-	0	3	8
Eclipse_jetty_4	✓						2 −	2 −	9	1	9	-	0	2	7
example_PWCheck	✓	✓	✓				1 ⊂	2 =	10	1	8	1	0	0	13
github_authmreloaded	✓	✓	✓				1 ⊥	4 ⊥	5	1	11	1	0	0	26
jasypt_digestEquals	✓	✓	✓				1 =	2 *D*	3	1	CE	1	0	CE	60
rsa_modpow_1717	✓	✓				✓	3 −	ID	49	1	ID	-	0	ID	6
rsa_modpow_1964903306	✓	✓				✓	3 −	ID	71	2	ID	-	0	ID	7
rsa_modpow_834443	✓	✓				✓	3 −	ID	69	2	ID	-	0	ID	6
shiro_hashEquals	✓	✓	✓				1 =	2 *D*	5	1	5	1	0	6	53
stac_ibasys	✓	✓			✓		2 −	1 −	9	9	9	-	0	0	6
themis_boot-stateless-auth	✓	✓	✓				1 =	1 *D*	33	2	15	2	0	0	20
themis_jdk	✓	✓	✓				1 ⊥	2 ⊥	2	1	1	1	0	0	6
themis_oacc	✓	✓	✓				1 −	2 −	12	1	17	-	0	2	13
themis_openmrs-core	✓						1 −	ID	2	1	ID	-	0	ID	25
themis_orientdb	✓	✓	✓				1 ⊥	3 ⊥	17	1	17	1	0	0	51
themis_picketbox	✓	✓	✓				1 ⊥	1 ⊥	17	1	17	1	0	2	10
themis_spring-security	✓	✓	✓				1 ⊂	2 =	2	1	16	1	0	2	5
tink_multiply	✓						1 =	ID	2	1	ID	1	0	ID	10
apache_ftpserver_md5	✓	✓		✓			2 ⊥	ID	7	6	ID	1	0	ID	21
apache_ftpserver_salted_encrypt	✓	✓					1 ⊥	ID	74	1	ID	76	-	-	-
apache_ftpserver_salted	✓	✓		✓			3 ⊥	ID	58	1	ID	60	-	-	-
blazer_gpt14	✓	✓				✓	4 ⊥	1 ⊥	69	13	63	25	4	0	15
blazer_k96	✓	✓				✓	3 ⊃	1 =	84	9	77	11	1	0	12
blazer_unixlogin		✓					1 =	ID	2	2	ID	2	1	ID	8
Eclipse_jetty_3	✓	✓				✓	3 −	3 −	24	2	23	-	1	3	7
themis_jetty	✓	✓	✓				2 ⊥	4 ⊥	14	17	20	39	1	0	8
themis_tomcat	✓	✓	✓				3 ⊃	3 ⊃	2	2	2	2	-	-	-
apache_ftpserver_stringutils	✓	✓	✓			✓	2 =	4 ⊃	17	CE	9	9	CE	0	38
blazer_loopandbranch	✓	✓					1 =	4 ⊃	2	IL	CE	2	IL	CE	17
blazer_sanity	✓	✓	✓				1 =	3 ⊃	2	IL	1	1	IL	1	24
themis_pac4j	✓	✓					1 =	ID	2	CE	ID	2	CE	ID	20
themis_pac4j_ext	✓	✓					1 =	1 =	2	CE	2	2	CE	0	21

[a]Op=unsafe operator; If=if statement; Ret/Cont/Brk=if statement also has early return/continue/break; Lop=loop statements.

[b]⊂: fix locations are a subset of the developer's. ⊃, superset; =, equal; *I*, intersecting; *D*, disjoint; −, dev fix not available; ⊥, dev fix alters semantics.

[c]ID=the repaired program is identical with the original program; CE=repair causes compilation error; IL=repair causes an infinite loop.

[d]Dash (-) indicates that a developer fix is not available.

[e]Dash (-) indicates that EvoSuite failed to generate a test suite that the original program can pass.

verification using SymDiff [35]. Finally, all patches are manually examined to not miss any side effects. For RQ1–3, the same evaluation is performed for DifFuzzAR. For RQ4, the time impact, we compare the execution time of the original program, the Pendulum-fixed program, and the developer fix, on the EvoSuite-generated test suite. For space impact, we count the number of lines of the patches. Note that we have chosen to compare with real-world developer fixes because they represent how well humans can repair vulnerabilities in a practical setting. Although these fixes may not be the only possible fix, the fact that they have been merged into real-world programs has shown

their acceptance in practice. Particularly, they show what amount of mitigation, corresponding performance degradation, and code change is acceptable for software developers.

Evaluation Setup. We first describe the setup for PENDULUM. For RQ1, as fuzzing is random, we run QFUZZ for three rounds, each 30 minutes long, for each program. Of all three runs, the set of secret values that reveal the most partitions is used to drive fix localization. For RQ2, for the repair of subjects with a loop statement, we provide the required loop bounds manually. The repaired programs are fuzzed with the same configuration as in RQ1; we report the greatest number of partitions throughout all rounds. With regard to DIFFUZZAR, we provide it with a required DIFFUZZ driver. The very drivers in its replication package are used whenever available; we make minimal modifications, if necessary, so that DIFFUZZAR has the same entry functions in the drivers as PENDULUM. For RQ4, to measure time impact, the test suite is executed with the JUNIT runner. Only the runtime spent in the program-to-fix counts; the runtime of JUNIT is excluded. Each execution is repeated 100 times, with just-in-time compilation disabled to capture the performance of the bytecode as it is. The average runtime and the standard deviation are reported. To measure patch size, the number of lines is counted for each file after they have been purged of all comments and reformatted with GOOGLE-JAVA-FORMAT; only non-blank lines count.

Environment and Configurations. We conducted all experiments on Ubuntu 16.04 LTS on an Intel Xeon E5-2660 v4 @ 2.00GHz machine with 62GB of memory. We use OPENJDK 1.8.0_292, GCC 5.5.0, EVOSUITE 1.2.0, JUNIT 4.12, and GOOGLE-JAVA-FORMAT 1.7. For QFUZZ, we use the same parameters as the original QFUZZ experiments.

5.3 Fix Localization (RQ1)

Table 1, *Fix Locations* column shows the number of fix locations identified as well as how they compare with the developer fix. For *all* subjects, PENDULUM (PDL) can identify at least one fix location. Exactly one fix location is identified for 28 out of 42 subjects, and at most four locations are identified for the others. In contrast, DIFFUZZAR (DFZ) fails to identify any fix location for 13 subjects. This is mainly because it tries to fix the entry method used by the fuzz driver, which is not always vulnerable. This leads it to produce "repairs" that are identical to the original programs (marked ID in Table 1), which retain all vulnerabilities.

A developer fix is available for 31 subjects. For 14 subjects, PENDULUM identifies the same locations as the developer fix. For three subjects, PENDULUM identifies a subset of the developer's fix locations but still achieves the same vulnerability mitigation. For two subjects, PENDULUM identifies a superset of fix locations. For the remaining 12 subjects, the developer fix changes the program logic altogether, e.g., changing string comparison to string hash comparison (themis_picketbox). This renders a comparison meaningless, as PENDULUM does not attempt logical changes.

The quality of our fix locations is further measured by PENDULUM's ability to mitigate the vulnerabilities at these locations. And indeed, the fix locations lead to significant mitigation of the vulnerabilities (as discussed in Section 5.4), indicating that our localization algorithm provides a solid basis for mitigating vulnerabilities.

RQ1—Fix Localization:
PENDULUM can perform fix localization successfully for all 42 subjects, while DIFFUZZAR fails for 13 subjects.

5.4 Vulnerability Mitigation (RQ2)

Table 1, SC *Partitions* column shows the number of partitions for the original program (Orig), the PENDULUM-generated fix (PDL), the DIFFUZZAR-generated fix (DFZ), and the developer fix (Dev).

Note that in our evaluation, we attempt to repair all identified locations without any selection or ranking. This is because, for most subjects (28 out of 42), there is only one identified location. Also, by attempting to repair all locations, we can observe the maximum possible vulnerability mitigation by Pendulum.

After being repaired by Pendulum, 33 of the 42 subjects show a reduced number of partitions. Among the 33 subjects, 26 end up with a *single* partition, i.e., the vulnerability has been eliminated. For four subjects, two partitions are found after the fix. For Eclipse_jetty_3, this is due to an out-of-bound access introduced by our repair. The other three subjects could have been completely fixed with the fix patterns; however, the relevant locations were not revealed by the set of secrets that led to the most partitions in the localization step. Still, as the major vulnerabilities are detected and addressed, the vulnerabilities in these three subjects are significantly mitigated. For complete vulnerability elimination, a user can apply Pendulum iteratively. In fact, it only takes QFuzz a few seconds to reveal the residual vulnerability in these three subjects. Another noteworthy subject is themis_jetty. Our repair seems to have increased the number of observed partitions from 14 to 17, but the original program also has 17 partitions, as presented in the results of QFuzz [45]. It is only due to the incompleteness of fuzzing that fewer partitions were revealed for the original program. Still, Pendulum fails to mitigate the vulnerability here, as the repair introduces out-of-bound string accesses. Such shortcomings can be tackled by, e.g., adding a safe utility method for array access.

In contrast, DifFuzzAR mitigates the vulnerability for only 15 subjects. Five of these are left with one partition, while the others are often left with a considerable number of partitions. Moreover, it fails to modify the programs of 13 subjects and exacerbates the vulnerability for four subjects.

RQ2—Vulnerability Mitigation:
Pendulum is able to mitigate the vulnerability effectively for 33 of 42 subjects. For 26 of these 33 subjects, Pendulum can entirely eliminate the SC vulnerability. In contrast, DifFuzzAR can mitigate the vulnerability for only 15 subjects.

5.5 Side Effect (RQ3)

We have performed both regression testing and formal verification to check the Pendulum-generated repairs for any side effect, i.e., change to the original program semantics. In Table 1, SC *Partitions* column, light shade indicates the presence of side effects, while dark shade indicates that the repair does not change the original program at all or prevents normal execution. Of the 42 repairs generated by Pendulum, 28 are free of side effects, among which 27 repairs reduce the number of partitions (to either one or two).

We have performed regression testing with EvoSuite, the state-of-the-art regression testing tool for Java. In Table 1, *Regression Test Failure* column shows the number of generated tests (*Total* column) and of failing tests. For those subjects for which EvoSuite was unable to find a failing regression test, we additionally performed a formal verification. We first used jar2bpl, a component of the Bixie [39] code checker, to translate the class files of the program before and after repair into BoogiePL [20], an intermediate verification language. We then verify the equivalence of the two program versions with SymDiff [35], an equivalence-checking tool. We have chosen this approach because BoogiePL is a widely used, well-established verification language. Also, the other state-of-the-art equivalence checking tools either lack support for Java [23, 25, 41, 57] or can only handle relatively simple program differences [6]. To perform the verification, we have written stubs for methods that jar2bpl does not model. For example, while jar2bpl models the primitive `int` type, it does not model the `Integer` wrapper class. Therefore, we write a stub for `Integer`, implementing its methods that are involved in the benchmark using the subset of Java

that is modeled by JAR2BPL. We also modify SymDiff with regard to heap comparison. Originally, SymDiff considers two methods equivalent only if they allocate *exactly* the same objects on the heap. We have modified SymDiff to allow for a more *relaxed* sense of equivalence. We consider two methods equivalent as long as all the objects allocated in the *pre-repair* version remain the same in the *post-repair* version. This definition allows that the post-repair method allocates some *additional* objects. This modification is made mainly with the `ite` utility method (see Listing 5) in mind. Since the `ite` method is generic, variables of primitive types (e.g., `int` and `boolean`) would be autoboxed into objects of the corresponding wrapper classes, so additional objects are allocated on the heap. However, the additional allocation does not change program semantics.

The main cause of changed semantics is impure method calls, within which object states are altered. Our fix pattern omits them and only takes care of explicit assignments, thus changing the semantics. In principle, though, as long as the program does not interact with the environment (e.g., database), impurity largely comes down to variable assignment, which can be made time-constant with our conditional assignment construct. Another cause is object access guarded by a null check, where our fix pattern can lead to a null pointer exception. Other failures include infinite loops (marked IL in Table 1) and out-of-bound array access, which may result when statements inside a loop are modified.

The DiffFuzzAR-generated repairs are also subjected to examination and regression testing. It is shown that only 11 repairs from DiffFuzzAR mitigate the vulnerability without side effect. Also, note that the mitigation is often minor, as in, e.g., blazer_gpt14 and blazer_k96.

RQ3—Side Effect:
28 out of 33 mitigating repairs from Pendulum are also side-effect-free, while DiffFuzzAR can only generate 11 such repairs, often providing minor vulnerability mitigation.

5.6 Time and Space Impact (RQ4)

For RQ4, we include only subjects that have a regression test suite as well as a side-effect-free, mitigating repair from Pendulum. We do not compare with DiffFuzzAR because it generates few such repairs.

Table 2, *Average Execution Time* column compares the runtime of the regression test suite between the original program, the developer fix, and the Pendulum-fixed program. Column $\Delta Orig$ shows the percentage difference to the original program, which is 43.0% on average. The slowdown is more significant where a loop is repaired (e.g., blazer_modpow2) and modest elsewhere; the median is thus only 2.6%. Column ΔDev shows an average slowdown of 0.4% compared with the developer fix. Furthermore, we want to emphasize that we are evaluating the *maximum* performance impact the repair has on the program because we repair all fix locations in our evaluation, as mentioned in Section 5.4. In practice, however, which and how many locations to repair remains a trade-off between security and performance that needs to be decided by the developer because it depends on the specific situation. Completely eliminating all SC partitions can potentially have a significant impact on the performance. Pendulum can be adjusted to mitigate only a subset of the identified locations, and the user can also explore different configurations to decide which amount of mitigation is suitable for their specific case. To illustrate this tradeoff, we selected four subjects from our benchmark, for which Pendulum identified multiple fix locations and successfully mitigated the vulnerability. We then use Pendulum to partially fix these subjects and measure the vulnerability reduction and the performance. The results are shown in Table 3. As more locations are fixed, these four subjects gradually have fewer SC partitions, while their execution time can

Table 2. Results for the Performance and Patch Size Comparison

Subject	Average Execution Time (msec)					Lines of Code				
	Orig	Dev	PDL	Δ Orig (%)	Δ Dev (%)	Orig	Dev	PDL	Δ Orig	Δ Dev
apache_ftpserver_clear	38 ± 9	51 ± 8	39 ± 10	2.6	-23.5	61	54	66	5	12
apache_wss4j	45 ± 11	-	46 ± 11	2.2	-	31	-	36	5	-
blazer_array	101 ± 13	101 ± 13	103 ± 16	2.0	2.0	61	63	60	-1	-3
blazer_login	16 ± 7	15 ± 6	15 ± 7	-6.3	0.0	21	28	28	7	0
blazer_modpow1	293 ± 17	358 ± 19	361 ± 19	23.2	0.8	128	130	128	0	-2
blazer_modpow2	23,941 ± 392	168,891 ± 80,126	174,218 ± 83,028	627.7	3.2	92	92	100	8	8
blazer_passwordEq	13 ± 7	13 ± 6	14 ± 4	7.7	7.7	20	20	26	6	6
blazer_straightline	174 ± 17	176 ± 16	182 ± 15	4.6	3.4	323	112	322	-1	210
Eclipse_jetty_1	16 ± 5	-	18 ± 10	12.5	-	8	-	15	7	-
Eclipse_jetty_2	17 ± 5	-	18 ± 5	5.9	-	9	-	10	1	-
Eclipse_jetty_4	17 ± 8	-	16 ± 6	-5.9	-	11	-	12	1	-
example_PWCheck	28 ± 9	28 ± 7	27 ± 7	-3.6	-3.6	35	41	42	7	1
github_authmreloaded	196 ± 17	196 ± 16	200 ± 17	2.0	2.0	65	54	70	5	16
rsa_modpow_1717	14 ± 6	-	20 ± 5	42.9	-	25	-	27	2	-
rsa_modpow_1964903306	16 ± 6	-	22 ± 7	37.5	-	26	-	28	2	-
rsa_modpow_834443	14 ± 7	-	18 ± 7	28.6	-	26	-	28	2	-
stac_ibasys	99 ± 12	-	98 ± 9	-1.0	-	62	-	63	1	-
themis_boot-stateless-auth	2,060 ± 36	2,058 ± 46	2,078 ± 50	0.9	1.0	87	89	94	7	5
themis_jdk	13 ± 7	13 ± 6	13 ± 5	0.0	0.0	12	13	19	7	6
themis_oacc	34 ± 8	-	35 ± 10	2.9	-	41	-	48	7	-
themis_openmrs-core	248 ± 16	-	250 ± 16	0.8	-	174	-	175	1	-
themis_orientdb	22,484 ± 58	111,166 ± 13,218	105,329 ± 6,218	368.5	-5.3	210	211	215	5	4
themis_picketbox	22 ± 11	21 ± 6	22 ± 6	0.0	4.8	23	18	30	7	12
themis_spring-security	12 ± 9	11 ± 3	11 ± 4	-8.3	0.0	28	25	33	5	8
jasypt_digestEquals	545 ± 37	545 ± 36	589 ± 38	8.1	8.1	359	365	366	7	1
shiro_hashEquals	940 ± 35	955 ± 32	944 ± 32	0.4	-1.2	148	146	153	5	7
tink_multiply	64 ± 12	62 ± 9	67 ± 11	4.7	8.1	19	19	22	3	3
Average	-	-	-	43.0	0.4	-	-	-	4.2	16.9
Median	-	-	-	2.6	1.0	-	-	-	5.0	6.0

The time measurement is performed on the regression tests.

Table 3. The Change of SC Partitions and Execution Time as More Locations Are Fixed

Subject	Side-Channel Partitions				Average Execution Time (msec)			
	Orig	PDL-1	PDL-2	PDL-3	Orig	PDL-1	PDL-2	PDL-3
Eclipse_jetty_4	9	2	1	-	17 ± 8	16 ± 4	16 ± 6	-
rsa_modpow_1717	49	39	21	1	14 ± 6	14 ± 3	14 ± 4	20 ± 5
rsa_modpow_1964903306	71	39	12	2	14 ± 7	14 ± 4	14 ± 3	18 ± 7
rsa_modpow_834443	69	62	15	2	16 ± 6	17 ± 3	17 ± 4	22 ± 5

PDL-1 refers to repairing 1 location using PENDULUM; similar for PDL-2 and PDL-3. Note that Eclipse_jetty_4 has only two fix locations and thus does not have PDL-3.

increase. Deciding on such a trade-off is eventually up to the user; PENDULUM produces source-level repairs, providing them with the means to make an informed decision.

In terms of space, or file size, PENDULUM implements the utility methods in 105 lines. Apart from this, PENDULUM increases the size of the file-to-fix. Table 2, *Lines of Code* column shows that PENDULUM's repair is on average 4.2 lines larger than the original program and 16.9 lines larger than the developer fix. Excluding blazer_straightline, where the developer fix completely changes the program semantics, PENDULUM's repair is on average six lines larger than the developer fix. Developer fixes can be small due to the use of additional library methods and occasional changes in the program semantics.

RQ4—Time and Space Impact:
The PENDULUM-generated repairs have an average slowdown of 43.0% and a median slowdown of 2.6%. This performance is close to that of the developer fixes. Our median repairs are five lines larger than the original code and six lines larger than the developer fixes.

6 Threats to Validity

External Validity. To mitigate the threat that our approach may not generalize, we use a wide range of micro-benchmarks and real-world Java programs for our experimental subjects. Specifically, they are an extended version of QFUZZ's benchmark [45], which is the state-of-the-art SC detection benchmark for Java. Additionally, there is a threat that our source-level repair gets subsumed by compiler optimizations [21], and further, that the compiler or interpreter itself introduces SC vulnerabilities [13]. However, our experiments show that the vulnerabilities are mitigated. Further, we intentionally target the source code to aid the developer's comprehension and fix the vulnerability where it is introduced, if it can be fixed in the source code at all.

Internal Validity. Our approach uses a fuzzing-based tool, QFUZZ [45]. To counter the randomness and incompleteness of fuzzing and obtain an accurate quantification of SC vulnerabilities, we repeat every fuzz campaign for three times, with the same (validated) setup reported in QFUZZ. The data obtained for the original programs are consistent with those reported in QFUZZ. To assess the possible side effects of the generated repairs, we corroborate our manual examination with a regression test performed with EVOSUITE [24]. While testing is incomplete, EVOSUITE is one of the state-of-the-art Java test case generation techniques and achieves high code coverage. For all repairs that pass the regression tests, we formally verified that they are semantically equivalent to the original program using SYMDIFF [35]. We also open-source our tool and all subjects, experimental setups, and repairs for public scrutiny.

Iterative Repair. In our experiments, we do not perform an iterative repair (as shown by the back edge from step four to step two in Figure 1) for ease of presentation. In practice, iterative detection and repair can help counter the incompleteness of fuzzing and incrementally reduce residual vulnerability. For example, the vulnerability of the subjects rsa_modpow_834443 and stac_ibasys can be eliminated with one more iteration.

7 Related Work

Refactoring. Program refactoring transforms the code to improve the system design while preserving the functionality [1]. While we also attempt to preserve the functionality, we aim to change the program's non-functional behavior, i.e., timing, to mitigate the SC vulnerability. Therefore, we regard our approach as program repair rather than program refactoring.

Synthesis. FACT [16] is a cryptographic domain-specific language that can be transformed into constant-time LLVM bitcode. Rather than synthesize new programs, we aim to repair existing JAVA programs.

Verification and Testing. Approaches have been proposed to verify the absence of timing SC vulnerability at source code level [4, 42, 48], intermediate representation (IR) level [3, 5, 18, 50], assembly level [2, 12] and binary level [22, 34]. There is also an approach for proving that the cryptographic constant-time property of a program is preserved during compilation [8]. Testing approaches generate concrete input to expose the presence of timing SC vulnerabilities. The input is generated with fuzz testing [9, 28, 43, 45] or symbolic execution [14, 17, 46]. The verification and testing approaches do not aim to mitigate the vulnerability but provide a basis for doing so. In

particular, our approach obtains a quantification of the vulnerability with QFuzz [45], which is then used to drive the fix localization.

Localization. CacheD [60] and SymSC [26] use symbolic execution and constraint solving to localize timing SCs, which are expensive computations. SC-Eliminator [61] uses static taint analysis to find secret-dependent instructions, which has a high false-positive rate because of overtainting. Fuchsia [56] uses a **decision tree (DT)** to explain SC partitions and takes the discriminants as leaky blocks. The DT cannot handle branch-induced SCs properly because it always blames some block in either branch rather than the branching block; in other words, it locates the effect rather than the cause. It also cannot distinguish between multiple blocks that always have the same execution counts. Besides, it is unclear how many discriminants to take from the DT as fix locations. Different from these works, our localization is computationally cheap, accurate, and handles different program constructs properly.

Repair. Debreach [47] mitigates timing SCs exclusively for compressor programs. Raccoon [49] and SC-Eliminator [61] repair timing SCs in LLVM IR, and hence, follow a different strategy than Pendulum. Our repair happens from the developer's perspective, providing them with fine-grained control to review fixes and trade between performance and security by carefully prioritizing the fixes. IR-based repair techniques are orthogonal to our approach and can be used if a source code-based repair is not feasible. DiffFuzzAR [37] is the closest to our work, which mitigates timing SCs for Java programs at the source level. It assumes that the vulnerable method is known, which may not be the case. Even when it works on the correct method, its fix patterns are far less effective than ours, as shown by our experiment.

8 Conclusion

In this article, we present a first approach for localizing timing SC vulnerabilities and mitigating them at source code automatically via program repair. Our tool Pendulum integrates a quantitative fuzzing technique with fault localization to transform SC vulnerable programs into safe ones using a set of repair patterns. We found that Pendulum can effectively mitigate SC vulnerabilities in a large set of real-world Java applications with minimal side effects and performance degradation. As a future direction, our approach of dynamic fix localization combined with pattern-based repair may be extended to other SC vulnerability types, e.g., cache-related timing vulnerabilities. Furthermore, the incompleteness of our dynamic analysis for SC detection can be mitigated by combining it with a more conservative static analysis, e.g., based on anti-patterns. We imagine that such a hybrid detection combined with our repair approach can achieve an even better overall framework.

References

[1] Chaima Abid, Vahid Alizadeh, Marouane Kessentini, Thiago do Nascimento Ferreira, and Danny Dig. 2020. 30 Years of Software Refactoring Research: A Systematic Literature Review. arXiv: 2007.02194. Retrieved from https://arxiv.org/abs/2007.02194

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, 1807–1823. DOI: https://doi.org/10.1145/3133956.3134078

[3] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 53–70. Retrieved from https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida

[4] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. 2013. Formal Verification of Side-Channel Countermeasures Using Self-Composition. *Science of Computer Programming* 78, 7 (7 2013), 796–812. DOI: https://doi.org/10.1016/j.scico.2011.10.008

[5] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. Decomposition Instead of Self-Composition for Proving the Absence of Timing Channels. In *Proceedings of the ACM-SIGPLAN Symposium on Programming Language Design and Implementation*, 362–375. DOI: https://doi.org/10.1145/3062341.3062378

[6] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. ARDiff: Scaling Program Equivalence Checking via Iterative Abstraction and Refinement of Common Code. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event) *(ESEC/FSE '20)*. ACM, New York, NY, 13–24. DOI: https://doi.org/10.1145/3368089.3409757

[7] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Păsăreanu, and Tevfik Bultan. 2016. String Analysis for Side Channels with Segmented Oracles. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. ACM, New York, NY, 193–204. DOI: https://doi.org/10.1145/2950290.2950362

[8] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time," In *31st IEEE Computer Security Foundations Symposium (CSF '18)*, 328–343. DOI: https://doi.org/10.1109/csf.2018.00031

[9] Tiyash Basu, Kartik Aggarwal, Chundong Wang, and Sudipta Chattopadhyay. 2020. An Exploration of Effective Fuzzing for Side-Channel Cache Leakage. *Software Testing, Verification and Reliability* 30, 1 (2020), e1718. DOI: https://doi.org/10.1002/stvr.1718

[10] Jan A. Bergstra and Alban Ponse. 2010. Short-Circuit Logic. arXiv: 1010.3674. Retrieved from http://arxiv.org/abs/1010.3674

[11] Elisa Bertino (Ed.). 2000. ECOOP 2000 - Object-Oriented Programming. In *Proceedings of 14th European Conference*, Lecture Notes in Computer Science, Vol. 1850. Springer, Berlin. DOI: https://doi.org/10.1007/3-540-45102-1

[12] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 917–934. Retrieved from https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond

[13] Tegan Brennan, Nicolás Rosner, and Tevfik Bultan. 2020. JIT Leaks: Inducing Timing Side Channels through Just-In-Time Compilation. In *2020 IEEE Symposium on Security and Privacy (SP '20)*, 1207–1222. DOI: https://doi.org/10.1109/sp40000.2020.00007

[14] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut T. Kandemir. 2019. CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation. In *2019 IEEE Symposium on Security and Privacy (SP '19)*, 505–521. DOI: https://doi.org/10.1109/sp.2019.00022

[15] Billy Bob Brumley and Nicola Tuveri. 2011. Remote Timing Attacks Are Still Practical. In *Proceedings of 6th European Symposium on Research in Computer Security (ESORICS '11)*. Vijay Atluri and Claudia Díaz (Eds.), Lecture Notes in Computer Science, Vol. 6879, 355–371. DOI: https://doi.org/10.1007/978-3-642-23822-2_20

[16] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: A DSL for Timing-Sensitive Computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, 174–189. DOI: https://doi.org/10.1145/3314221.3314605

[17] Sudipta Chattopadhyay. 2017. Directed Automated Memory Performance Testing. In *23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '17)*, Held as Part of the European Joint Conferences on Theory and Practice of Software *(ETAPS '17)*, Proceedings, Part II. Axel Legay and Tiziana Margaria (Eds.), Lecture Notes in Computer Science, Vol. 10206, 38–55.

[18] Jia Chen, Yu Feng, and Isil Dillig. 2017. Precise Detection of Side-Channel Vulnerabilities Using Quantitative Cartesian Hoare Logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, 875–890. DOI: https://doi.org/10.1145/3133956.3134058

[19] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. 2010. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *31st IEEE Symposium on Security and Privacy (S & P '10)*, 191–206. DOI: https://doi.org/10.1109/sp.2010.20

[20] R. Deline and K. Leino. 2005. *BoogiePL: A Typed Procedural Language for Checking Object-Oriented Programs*. Technical Report. Citeseer.

[21] Chaoqiang Deng and Kedar S. Namjoshi. 2016. Securing a Compiler Transformation. In *International Static Analysis Symposium*, Lecture Notes in Computer Science, Vol. 9837,170–188. DOI: https://doi.org/10.1007/978-3-662-53413-7_9

[22] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2015. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. *ACM Transactions on Information and System Security* 18, 1 (6 2015), Article 4, 1–32. DOI: https://doi.org/10.1145/2756550

[23] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating Regression Verification. In *Proceedings of ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.), 349–360. DOI: https://doi.org/10.1145/2642937.2642987

[24] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2 2013), 276–291. DOI: https://doi.org/10.1109/tse.2012.14

[25] Benny Godlin and Ofer Strichman. 2013. Regression Verification: Proving the Equivalence of Similar Programs. *Software Testing, Verification and Reliability* 23, 3 (5 2013), 241–258.

[26] Shengjian Guo, Meng Wu, and Chao Wang. 2018. Adversarial Symbolic Execution for Detecting Concurrency-Related Cache Timing Leaks. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE 2018)*. Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.), 377–388. DOI: https://doi.org/10.1145/3236024.3236028

[27] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. 2006. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006)*, 57–66. https://doi.org/10.1109/rtss.2006.12

[28] Shaobo He, Michael Emmi, and Gabriela F. Ciocarlie. 2020. ct-fuzz: Fuzzing for Timing Leaks. In *Proceedings of the 13th IEEE International Conference on Software Testing, Validation and Verification, (ICST '20)*, 466–471. DOI: https://doi.org/10.1109/icst46399.2020.00063

[29] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic Fuzzer Generation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2271–2287. Retrieved from https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou

[30] Sharjeel Khan, Girish Mururu, and Santosh Pande. 2020. A Compiler Assisted Scheduler for Detecting and Mitigating Cache-Based Side Channel Attacks. arXiv.org: 2003.03850. Retrieved from https://arxiv.org/abs/2003.03850

[31] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP '19)*, 1–19. DOI: https://doi.org/10.1109/sp.2019.00002

[32] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems., In *Proceedings of the 16th Annual International Cryptology Conference (CRYPTO 96)*, Neal Koblitz (Ed.), Lecture Notes in Computer Science, Vol. 1109, 104–113. DOI: https://doi.org/10.1007/3-540-68697-5_9

[33] Boris Köpf and David Basin. 2007. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, New York, NY, USA, 286–296. DOI: https://doi.org/10.1145/1315245.1315282

[34] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2012. Automatic Quantification of Cache Side-Channels. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV '12)*, P. Madhusudan and Sanjit A. Seshia (Eds.), Lecture Notes in Computer Science, Vol. 7358, 564–580. DOI: https://doi.org/10.1007/978-3-642-31424-7_40

[35] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV '12)*, P. Madhusudan and Sanjit A. Seshia (Eds.), Lecture Notes in Computer Science, Vol. 7358, 712–717. DOI: https://doi.org/10.1007/978-3-642-31424-7_54

[36] Nate Lawson. 2009. Timing Attack in Google Keyczar Library. Retrieved from https://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/

[37] Rui Lima, João F. Ferreira, and Alexandra Mendes. 2021. Automatic Repair of Java Code with Timing Side-Channel Vulnerabilities. In *36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW '21)*, 1–8. DOI: https://doi.org/10.1109/asew52652.2021.00014

[38] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-Based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*. ACM, New York, NY, 31–42. DOI: https://doi.org/10.1145/3293882.3330577

[39] Tim McCarthy, Philipp Rümmer, and Martin Schäf. 2015. Bixie: Finding and Understanding Inconsistent Code. In *37th IEEE/ACM International Conference on Software Engineering (ICSE '15)*, 645–648. DOI: https://doi.org/10.1109/icse.2015.213

[40] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2002. Parameterized Object Sensitivity for Points-to and Side-Effect Analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*. ACM, New York, NY, 1–11. DOI: https://doi.org/10.1145/566172.566174

[41] Federico Mora, Yi Li, Julia Rubin, and Marsha Chechik. 2018. Client-Specific Equivalence Checking. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. ACM, New York, NY, 441–451. DOI: https://doi.org/10.1145/3238147.3238178

[42] Van Chan Ngo, Mario Dehesa-Azuara, Matthew Fredrikson, and Jan Hoffmann. 2017. Verifying and Synthesizing Constant-Resource Implementations with Types. In *2017 IEEE Symposium on Security and Privacy (SP '17)*, 710–728. DOI: https://doi.org/10.1109/sp.2017.53

[43] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. 2019. DiffFuzz: Differential Fuzzing for Side-Channel Analysis. In *2019 International Conference on Software Engineering (ICSE '19)*, 176–187. DOI: https://doi.org/10.1109/icse.2019.00034

[44] Yannic Noller, Corina S. Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. 2020. HyDiff: Hybrid Differential Software Analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. ACM, New York, NY, 1273–1285. DOI: https://doi.org/10.1145/3377811.3380363

[45] Yannic Noller and Saeid Tizpaz-Niari. 2021. QFuzz: Quantitative Fuzzing for Side Channels. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual) *(ISSTA '21)*. ACM, New York, NY, 257–269. DOI: https://doi.org/10.1145/3460319.3464817

[46] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. 2016. Multi-Run Side-Channel Analysis Using Symbolic Execution and Max-SMT. In *IEEE 29th Computer Security Foundations Symposium (CSF '16)*, 387–400. DOI: https://doi.org/10.1109/csf.2016.34

[47] Brandon Paulsen, Chungha Sung, Peter A. H. Peterson, and Chao Wang. 2019. Debreach: Mitigating Compression Side Channels via Static Analysis and Transformation. In *34th International Conference on Automated Software Engineering (ASE '19)*, 899–911. DOI: https://doi.org/10.1109/ase.2019.00088

[48] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified Low-Level Programming Embedded. In *Proceedings of the ACM on Programming Languages* 1, ICFP (2 2017), Article 17, 1–17. DOI: https://doi.org/10.1145/3110261

[49] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, 431–446. Retrieved from https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/rane

[50] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. 2016. Sparse Representation of Implicit Flows with Applications to Side-Channel Detection. In *Proceedings of the 25th International Conference on Compiler Construction (CC '16)*. ACM, New York, NY, 110–120. DOI: https://doi.org/10.1145/2892208.2892230

[51] Atanas Rountev. 2004. Precise Identification of Side-Effect-Free Methods in Java. In *Proceedings of the* 20th *IEEE International Conference on Software Maintenance (ICSM '04)*, 82–91. DOI: https://doi.org/10.1109/icsm.2004.1357793

[52] Alexandru Salcianu and Martin C. Rinard. 2005. Purity and Side Effect Analysis for Java Programs. In *6th International Conference on Verification, Model Checking, and Abstract Interpretation, (VMCAI '05)*, Proceedings, Radhia Cousot (Ed.), Lecture Notes in Computer Science, Vol. 3385, 199–215. DOI: https://doi.org/10.1007/978-3-540-30579-8_14

[53] Thomas Sewell, Felix Kam, and Gernot Heiser. 2016. Complete, High-Assurance Determination of Loop Bounds and Infeasible Paths for WCET Analysis. In *Proceedings of the 2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 185–195. DOI: https://doi.org/10.1109/rtas.2016.7461326

[54] Geoffrey Smith. 2009. On the Foundations of Quantitative Information Flow. In *12th International Conference on* Foundations of Software Science and Computational Structures *(FOSSACS '09)*, Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2009), Proceedings, Luca de Alfaro (Ed.), Lecture Notes in Computer Science, Vol. 5504, 288–302. DOI: https://doi.org/10.1007/978-3-642-00596-1_21

[55] Saeid Tizpaz-Niari, Pavol Cerný, and Ashutosh Trivedi. 2019. Quantitative Mitigation of Timing Side Channels. In *31st International Conference on Computer Aided Verification (CAV '19)*, Proceedings, Part I, Isil Dillig and Serdar Tasiran (Eds.), Lecture Notes in Computer Science, Vol. 11561, 140–160. DOI: https://doi.org/10.1007/978-3-030-25540-4_8

[56] Saeid Tizpaz-Niari, Pavol Cerný, and Ashutosh Trivedi. 2020. Data-Driven Debugging for Functional Side Channels. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS '20)*. DOI: https://doi.org/10.14722/ndss.2020.24269

[57] Anna Trostanetski, Orna Grumberg, and Daniel Kroening. 2017. Modular Demand-Driven Analysis of Semantic Difference for Program Versions. In *Proceedings of the 24th International Static Analysis Symposium (SAS '17)*. Francesco Ranzato (Ed.), Lecture Notes in Computer Science, Vol. 10422, 405–427. DOI: https://doi.org/10.1007/978-3-319-66706-5_20

[58] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2021. oo7: Low-Overhead Defense Against Spectre Attacks via Program Analysis. *IEEE Transactions on Software Engineering* 47, 11 (7 2021), 2504–2519. DOI: https://doi.org/10.1109/tse.2019.2953709

[59] Jingbo Wang, Chungha Sung, and Chao Wang. 2019. Mitigating Power Side Channels during Compilation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*. ACM, New York, NY, 590–601. DOI: https://doi.org/10.1145/3338906.3338913

[60] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. 2017. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 235–252. Retrieved from https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-shuai

[61] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating Timing Side-Channel Leaks Using Program Repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '18)*. ACM, New York, NY, 15–26. DOI: https://doi.org/10.1145/3213846.3213851
[62] Michal Zalewski. 2017. American Fuzzy Lop. Retrieved from https://lcamtuf.coredump.cx/afl/
[63] Mingrui Zhang, Jianzhong Liu, Fuchen Ma, Huafeng Zhang, and Yu Jiang. 2021. IntelliGen: Automatic Driver Synthesis for Fuzz Testing. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 318–327. DOI: https://doi.org/10.1109/icse-seip52600.2021.00041

Received 15 February 2023; revised 19 June 2024; accepted 19 June 2024