JVM Fuzzing for JIT-Induced Side-Channel Detection*

Tegan Brennan
University of California Santa Barbara
Santa Barbara, CA, USA
tegan@cs.ucsb.edu

Seemanta Saha
University of California Santa Barbara
Santa Barbara, CA, USA
seemantasaha@cs.ucsb.edu

Tevfik Bultan
University of California Santa Barbara
Santa Barbara, CA, USA
bultan@cs.ucsb.edu

ABSTRACT

Timing side channels arise in software when a program's execution time can be correlated with security-sensitive program input. Recent results on software side-channel detection focus on analysis of program's source code. However, runtime behavior, in particular optimizations introduced during just-in-time (JIT) compilation, can impact or even introduce timing side channels in programs. In this paper, we present a technique for automatically detecting such JIT-induced timing side channels in Java programs. We first introduce patterns to detect partitions of secret input potentially separable by side channels. Then we present an automated approach for exploring behaviors of the Java Virtual Machine (JVM) to identify states where timing channels separating these partitions arise. We evaluate our technique on three datasets used in recent work on side-channel detection. We find that many code variants labeled "safe" with respect to side-channel vulnerabilities are in fact vulnerable to JIT-induced timing side channels. Our results directly contradict the conclusions of four separate state-of-the-art program analysis tools for side-channel detection and demonstrate that JIT-induced side channels are prevalent and can be detected automatically.

ACM Reference Format:
Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2020. JVM Fuzzing for JIT-Induced Side-Channel Detection. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380432

1 INTRODUCTION

Side-channel vulnerabilities occur when a non-functional but observable behavior of a system (such as its execution time) leaks information about the secret values that the system accesses. Although side-channel vulnerabilities due to hardware (such as vulnerabilities that exploit the cache behavior [13, 36, 40, 51]) have been extensively studied [4, 5, 32, 35], software side channels have only recently become an active area of research [9, 10, 47, 48, 55]. In recent years, several techniques have been proposed for detecting side-channel vulnerabilities in programs. In this paper, we demonstrate that the most recent analysis techniques and tools [7, 12, 17, 39] proposed for detection of side channels in Java programs miss a class of vulnerabilities and incorrectly label programs safe.

The main cause of this failure is that side-channel detection tools do not take dynamic behavior of the Java Virtual Machine (JVM) into account. Runtime performance of Java programs significantly depends on Just-In-Time (JIT) compilation techniques which compile and optimize portions of the code based on the runtime behavior of the program. In this paper, we demonstrate that programs labeled safe with respect to timing side channels by four different program analysis tools do in fact contain side-channel vulnerabilities if the runtime behavior of the program is sufficiently biased. We call side-channel vulnerabilities that are due to JIT optimizations JIT-induced side channels, and we present an automated approach for finding JIT-induced side-channel vulnerabilities.

Our contributions in this paper are:

- An automated pattern-based approach for finding input partitions that are likely to be separable by a timing side channel.
- An automated technique to generate input belonging to different partition cells using branch instrumentation.
- An automated search strategy for a JVM state vulnerable to a timing channel using priming input generated via fuzzing.
- An evaluation of our approach on widely-used side-channel detection benchmarks, demonstrating its ability to automatically induce timing side channels in programs labelled safe by four other recent analysis tools: Blazer [7], Themis [17], CoCo-Channel [12], and DiffFuzz [39].

The rest of the paper is organized as follows. In Section 2, we discuss JIT-induced timing side channels and provide an overview of our automated detection process. In Section 3, we discuss our approach for pattern-based identification of partitions of secret input which are likely to be separable by a timing side channel. We detail how we instrument the program to generate test input data belonging to these partition cells. We also discuss how we use grey-box fuzzing to generate a set of priming inputs. In Section 4, we present an algorithm to use the generated test input data and priming input to fuzz the JVM for vulnerabilities to JIT-induced timing side channels. In Section 5, we evaluate our technique, discuss our experiments and highlight key results. In Section 6, we discuss related work. In Section 7, we present our conclusions.

*This material is based on research sponsored by NSF under grants CCF-1817242 and CCF-1901098 and by DARPA under the agreement number FA8750-15-2-0087. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

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 ACM 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.
ICSE '20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380432

```
public static boolean sanity_unsafe(int a, int b) {
    int i = b;
    int j = b;
    if (b<0) return false;
    if (a<0) {
        return true;
    } else {
        while (i > 0) {
            i--;
        }
    }
        return false;
}
```

(a)

```
public static boolean sanity_safe(int a, int b) {
    int i = b;
    int j = b;
    if (b<0) return false;
    if (a>0) {
        while (i > 0) {
            i--;
        }
    } else {
        while (j > 0) {
            j--;
        }
    }
        return false;
}
```

(b)

Figure 1: Two examples from the Blazer benchmark.

2 OVERVIEW

Timing side-channel detection techniques investigate the following question: Assume a program p is a function on some input I consisting of secret input h and non-secret input l. For some fixed l, are there at least two mutually exclusive non-empty subsets of the secret domain h such that by observing the execution time of the program on a secret value, one can determine to which of the subsets the secret belongs? Note that precisely determining the answer to this membership question may not be possible due to either noise in the observations of the execution time or to overlapping execution time behavior of secret values belonging to different subsets of the input. So, typically, the answer we get to the question of "Does the secret belong to a particular subset?" is not a discrete "Yes" or "No" answer, but a quantitative answer that indicates our assessment of how likely it is that the secret belongs to that particular subset.

To automate the analysis we have to frame the problem in more detail. First, we need to characterize what type of information about the secret can be leaked from the program. At a high level, the type of information that can be leaked about the secret will characterize the subsets of the secret domain for which the membership question is most likely answerable. One avenue through which information can be leaked is program branches. Each branch in the program can potentially influence the execution time. Therefore, the branches that depend on the secret can result in timing side channels.

Consider a secret-dependent branch and the corresponding branch condition, and assume that we divide the set of secret values to two subsets: 1) The secret values that cause the branch condition to evaluate to true, 2) The secret values that cause the branch condition to evaluate to false. Now, also assume that by observing the execution time, we can tell if the branch condition evaluates to true or not (this would be possible, for example, if the evaluation of the branch condition to true causes more expensive computation to be undertaken). In this case, we can conclude that there is a timing side channel and the program leaks information about the secret since we are able to distinguish which subset that the secret belongs to.

Let us make our discussion more concrete with an example. Figure 1(a) shows the *sanity_unsafe* method taken from the Blazer side-channel detection benchmark [7]. In this example the secret is the value of the argument a. The second if statement in the program corresponds to a branch that is dependent on the secret. If argument b is particularly large, it would be possible to detect if the then or else-branch is taken. This would enable us to tell if the secret value is less than zero. So, we can define two subsets of the secret domain (integer values less than zero, and integer values greater than or equal to zero), and by observing the execution time of the method we can tell which subset the secret belongs to. Hence, we can conclude that this method has a side-channel vulnerability and is unsafe. And, this is exactly what the side-channel detection tools report. Profiling the execution time of the *sanity_unsafe* method shows a clear timing side-channel. The violin plot in Figure 2(a) shows the execution time distributions for the *sanity_unsafe* method for partitions: $a < 0$ and $a \geq 0$ for a fixed b value.

Now, let us look at the *sanity_safe* method in Figure 1(b). Again, the secret is the value of the argument a, and the second if statement in the program corresponds to a branch that is dependent on the secret. However, for this case, both branches contain equivalent computations, so the same computation is performed regardless of the evaluation of the branch condition. This should imply that we cannot tell which branch is taken by observing the execution time.

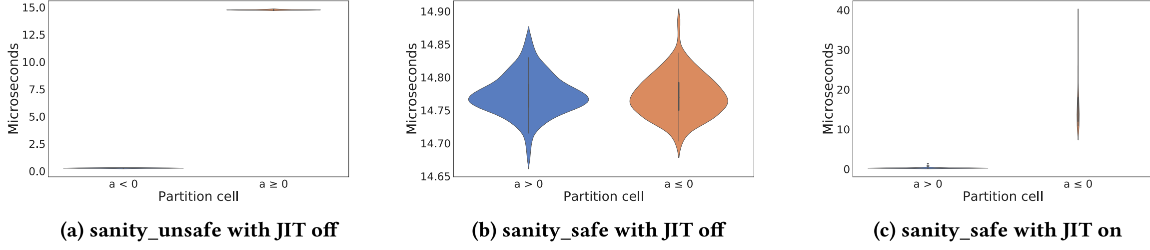
All modern side-channel analysis tools [7, 12, 17, 39] perform this reasoning in one form or another and conclude that the method *sanity_safe* is indeed safe and does not contain a side-channel vulnerability. Let us again check this conclusion by profiling. Figure 2(b) demonstrates the execution time distribution for the *sanity_safe* method for partitions $a > 0$ and $a \leq 0$ with the same constant value for b as used in Figure 2(a). It is clear from these distributions that an attacker cannot determine if the secret value is greater than zero or not by monitoring the execution time of the *sanity_safe* method since the timing behaviors for two cases are very similar, and, hence, there is no side-channel.

Unfortunately, this conclusion is wrong! The *sanity_safe* method shown in Figure 1(b) does contain a timing side-channel vulnerability. To understand why, we need to realize that the source code of *sanity_safe* is not the only factor on its execution time. The runtime environment itself plays a pivotal role in the program's behavior.

JIT-induced Side-Channels. The runtime environment can introduce timing channels into deceptively secure-looking programs as it attempts to optimize paths that it deems "hot" [11]. Just-In-Time (JIT) compilation is performed dynamically based on runtime behavior. The JIT compiler generates optimized native code so that the most commonly executed paths in a program execute as fast as possible. The result is that the prior input distribution to a program impacts its execution time on new input. Hence, if a program is called repeatedly in a way that causes the JIT compiler to optimize a particular execution path, then calls to the program on unknown input can leak information about what path that input follows.

The profiling data we show in Figure 2(b) was taken with JIT-disabled. If we enable JIT and then execute the *sanity_safe* method with a biased distribution where $a > 0$ is highly favored, JIT compilation will optimize this case. This leads to more efficient execution time for inputs where $a > 0$ and introduces a timing side channel leaking information about whether $a > 0$ or $a \leq 0$.

We profile again to determine if this hypothesis holds. Figure 2(c) shows the execution time distribution for the *sanity_safe* method again for the partitions: $a > 0$ and $a \leq 0$. We obtained these distributions by enabling JIT and repeatedly executing *sanity_safe* with an input distribution heavily favoring values where $a > 0$. After this priming stage, we timed calls to *sanity_safe* on input



(a) sanity_unsafe with JIT off (b) sanity_safe with JIT off (c) sanity_safe with JIT on

Figure 2: Execution time distributions for the sanity_safe and sanity_unsafe methods.

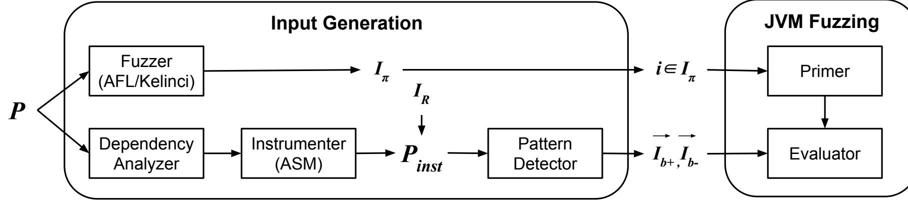


Figure 3: Automated JIT-Induced Side-Channel Detection

values from different partition cells and created a violin plot of the resulting timing distributions. We again kept the value of the public input b constant. The timing side channel is apparent from the clearly separable distributions for the two partition cells.

Currently, all side-channel analysis detection tools miss this vulnerability. This is due to its inception in the runtime behavior and in the biased input distribution provoking the JVM into favoring some program paths. Currently, no side-channel analysis tool can detect this class of side-channel vulnerabilities. A bias in the input distribution can arise for a variety of reasons. If an attacker can call a method repeatedly, then the attacker can force the JIT compiler to optimize a particular execution path going through a secret dependent branch. There might also be a natural bias in the input distribution. For example, if it is the case that most users call the *sanity_safe* method with a secret value $a > 0$ (or visa versa), then a JIT-induced side channel can arise based on this common behavior.

Ultimately, the key factor in forcing JIT compilation to induce a side channel is bias in the input distribution. This is a dynamic characteristic and the same program can be fed inputs in a countless variety of ways. We use the term *priming* to mean the act of executing a program repeatedly with a certain distribution of program input. Three factors determine a priming strategy: the program input used, the number of program calls made, and the bias in the input distribution. This huge search space of possible priming strategies calls for a systematic strategy of exploration if we have any hope of automatically detecting a program's vulnerability to JIT-induced side channels.

Automating JIT-Induced Side Channel Detection. Our automated approach consists of two main phases, outlined in Figure 3. The first phase is the Input Generation phase. This phase has two end goals. Given a program p taking secret input, we first will generate pairs of partition cells (I_{b^+}, I_{b^-}), where each cell is a subset of the secret domain. These pairs of partition cells are candidates of mutually exclusive non-empty subsets of the secret domain that may be distinguishable via a JIT-induced side channel.

In order to generate these partition cells, we define six program behavior patterns to identify input partitions potentially prone to side-channel vulnerabilities. We use static dependency analysis to identify secret dependent branches to use in these patterns. Then, we create an instrumented version of the program p_{inst}, introducing two counters for each branch. The first counts how many times the branch condition is reached. The second counts how many times branch condition evaluates to true. We generate a set of inputs I_R, run p_{inst} on these inputs and collect information for each input based on the branch counter values. We match these values to predefined behavior patterns which partition I_R to yield the desired (I_{b^+}, I_{b^-}) pairs. The lower half of the Input Generation block of Figure 3 overviews this generation of (I_{b^+}, I_{b^-}) evaluation pairs.

The second goal of the Input Generation phase is to generate a set of priming inputs I_π. Values from I_π are used to prime the JVM to favor certain program paths in an attempt to introduce timing side channels. We use the grey-box fuzzer AFL [53] to generate inputs executing different paths through the program. Input values that cover different program paths enable us to search different ways of priming the JVM while searching for a JVM state vulnerable to timing side channels.

The completion of the Input Generation phase yields a set of priming inputs I_π and a set of pairs of test partition cells $(\vec{I}_{b^+}, \vec{I}_{b^-})$.

The following JVM Fuzzing phase answers the question: For any given pair $(I_{b^+}, I_{b^-}) \in (\vec{I}_{b^+}, \vec{I}_{b^-})$ of input partition cells, does priming the JVM in favor of some input $i_\pi \in I_\pi$ result in a timing side channel through which input from I_{b^+} and I_{b^-} are distinguishable by their execution time? To answer, we systematically explore different priming strategies by iterating over different priming amounts and priming ratios. We evaluate if we successfully induced a timing side channel by quantifying the information leakage.

3 INPUT GENERATION

Below we discuss generation of input partitions and priming inputs.

3.1 Secret-Dependent Branch Detection

The first step of our approach is to identify secret dependent branches. A branch is secret dependent if the evaluation of the branch condition depends on the value of a secret program input. Given a program and a set of inputs marked as secret, we use static dependency analysis to identify the secret dependent branches.

We used Janalyzer [8], an existing static analysis tool, to perform this dependency analysis. Janalyzer constructs the system dependence graph (SDG) [29] for a given program. A SDG is constructed by first extracting the call graph for the program and the procedure dependence graph (PDG) [25] for each procedure. Nodes of the PDG are either statements or branch conditions and edges represent dependencies. A data dependence is a triple (d, u, v) calculated using reaching definitions analysis [38] where v is a variable and d and u are PDG nodes, v is defined in node d, used in node u, there is a path from d to u, and v is not redefined in between. For example, for the *sanity_unsafe* procedure shown in Figure 1(a), one such data dependence triple is (*argument a*, *if (a<0)*, *a*). Using SDG and PDGs, we do forward dependency analysis from all the marked secret program inputs and identify the program statements and branch conditions dependent on those inputs. We filter out all other statements keeping only the branch conditions as we are concerned about secret dependent branches only.

In our experiments, we observed that using both data and control dependencies introduces too many spurious dependencies. Hence, we consider only data dependencies during PDG construction but track both explicit (direct data flow using reaching definitions analysis) and implicit flows (data flows to v occurs inside a branch whose predicate is dependent on secret) following [17]. The end result is a set of secret-dependent branches.

Removing control dependencies leads to unsoundness in the dependency analysis, and our side-channel detection technique can generate false negatives, in the sense that if we are unable to detect a JIT-induced side-channel, that does not guarantee that the program is free of JIT-induced side channels. On the other hand, our side-channel detection technique does not generate false positives, i.e., when we do report a JIT-induced side channel, it means that there is demonstrable side channel in the program.

Although ignoring control dependencies reduces the number of spurious dependencies, data flow analysis can still lead to spurious dependencies. However, since we actually execute the program on input values we generate, these will not lead to any false reports of JIT-induced side channels.

3.2 Pattern Detection

A timing side channel is present if there are at least two mutually exclusive subsets of the secret domain such that an attacker who is able to observe the execution time of the program can determine which of the two subsets an unknown secret value belongs to (under the constraint that the secret does belong to one of the two). It is not feasible to explore the entire set of possible partitions of the secret domain to look for such subsets, and randomly chosen partitions are unlikely to yield meaningful results. To reduce the space of potential partition choices, we observe that timing side channels typically arise due to secret-dependent branches whose evaluation leads to an observable difference in the execution time.

We identify six program behavior patterns with respect to branch conditions corresponding to if statements, and use these patterns to define partitions of secret input. Matches to these patterns can be detected via instrumentation as we describe later in this section. One can define more patterns and corresponding partitions, however, our experiments demonstrate that these six are rich enough to effectively detect JIT-induced side-channel vulnerabilities.

Let I denote the set of all inputs for a given program p, where, given an input $i \in I$, $p(i)$ denotes the run that corresponds to executing program p on input i. We use h to denote the secret part of the input and l to denote the non-secret (public) part. Given an input $i \in I$ to the program, we write $i = \langle l, h \rangle$ to denote that the input consists of concatenation of public and secret parts. Given a program p, let b_p denote the set of branches in the program and $b_h \subseteq b_p$ denote the set of branches with branch conditions that are dependent on the secret h.

For any run $p(i)$ of the program and any $b \in b_p$, let $\#(p(i), b)$ be the total number of times the branch b is reached during the run $p(i)$ (i.e., the number of times the condition guarding b is evaluated during the run $p(i)$). Let $\#(p(i), b^+)$ denote the number of times the branch condition for b evaluates to true and $\#(p(i), b^-)$ denote the number of times the branch condition for b evaluates to false. Note that, for all p, b, and i, $\#(p(i), b) = \#(p(i), b^+) + \#(p(i), b^-)$.

Given a set of inputs I, our goal is to partition I using a secret-dependent branch $b \in b_h$. Given a branch $b \in b_h$, we partition I into I_{b^+}, I_{b^-} and I_0 where $I = I_{b^+} \cup I_{b^-} \cup I_0$ and I_{b^+}, I_{b^-} and I_0 are mutually exclusive. Below, we define six patterns we use for obtaining I_{b^+}, I_{b^-} and I_0.

We first consider the following pattern: $b \in b_h$ corresponds to an if statement which is not in a loop or a recursive function and can be reached at most once per each run of the program. We call this *Single Visit (SV)* branch pattern. For the SV pattern, we define I_{b^+}, I_{b^-} and I_0 as follows:

(1) $\forall i \in I_{b^+} \cup I_{b^-}, \#(p(i), b) = 1$.
(2) $\forall i \in I_{b^+}, \#(p(i), b^+) = 1$.
(3) $\forall i \in I_{b^-}, \#(p(i), b^-) = 1$.
(4) $I_{b^+} \neq \emptyset \wedge I_{b^-} \neq \emptyset$.
(5) $\forall i_1, i_2 \in I_{b^+} \cup I_{b^-}, \forall b_1 \in b_h \cup b_l, b_1 \neq b$,
$\#(p(i_1), b_1) = \#(p(i_2), b_1) \wedge \#(p(i_1), b_1^+) = \#(p(i_2), b_1^+)$.

Intuitively, this partition divides the set of inputs I into those that generate runs in which b is reached and the branch condition evaluates to true (I_{b^+}, (2)) and those that generate runs in which b is reached and branch condition evaluates to false (I_{b^-}, (3)) subject to the condition that all these runs agree on all other branch condition

evaluations (5). We also require I_{b^+} and I_{b^-} to be non-empty (4). All inputs that cannot be put into I_{b^+} or I_{b^-} are put into I_0.

As an example, for the *sanity_unsafe* procedure shown in Figure 1(a), using the SV pattern we obtain the following partition of the input domain: $I_{b^+} = \{a \mid a < 0\}$, $I_{b^-} = \{a \mid a \geq 0\}$ and $I_0 = \emptyset$,

I_{b^+} and I_{b^-} define two subsets of the secret domain whose separability via timing observations we wish to evaluate. Note that multiple partitions can satisfy the above criteria depending on the behavior of the runs that are generated by the inputs in I_{b^+} and I_{b^-} regarding other conditional branches. We can explore different partitions and if we can find one partition in which membership to I_{b^+} or I_{b^-} can be determined by timing observations, then we can conclude that there is a timing side-channel.

The requirement that runs generated by inputs in I_{b^+} and I_{b^-} must agree on all branch condition evaluations barring those on b itself is strong. It is possible that there might be no such partition cells I_{b^+} and I_{b^-}. For example, the evaluation of b to true could generate runs that reach a branch condition unreachable should b evaluate to false. This motivates our *Single Visit Relaxed (SVR)* branch pattern. Under this pattern, I_{b^+} and I_{b^-} are defined using the same rules for the SV pattern, except the rule (5) is modified as:

$\forall i_1, i_2 \in I_{b^+} \cup I_{b^-}, \forall b_1 \in b_h \cup b_l, b_1 \neq b,$
$(\#(p(i_1), b_1) > 0 \wedge \#(p(i_2), b_1) > 0) \Rightarrow$
$(\#(p(i_1), b_1) = \#(p(i_2), b_1) \wedge \#(p(i_1), b_1^+) = \#(p(i_2), b_1^+))$.

Now, runs generated from I_{b^+} and I_{b^-} need only agree on all branch evaluations reached by every run. Through another lens, this means that all runs share a common, possibly empty prefix and common, possibly empty suffix. In practice, we look for partitions that maximize the size of this shared prefix and suffix.

Next, let us consider a branch $b \in b_h$ corresponding to an if statement where b is either in a loop or in a recursive function. Hence, b might be reached multiple times. Again, there are many ways to partition the input domain based on such a branch condition. We will focus on four patterns: 1) *Multiple Visit All True (MVAT)* and 2) *Multiple Visit All False (MVAF)* and the relaxed variants of these two patterns 3) *Multiple Visit All True Relaxed (MVATR)* and 4) *Multiple Visit All False Relaxed (MVAFR)* which we define below.

In the MVAT pattern the branch b is visited multiple times. I_{b^+} consists of input values that generate runs where every time b is reached, the branch condition evaluates to true. I_{b^-} consists of input values that generate runs where at least one time when b is reached, the branch condition evaluates to false. We formalize the MVAT pattern as follows:

(1) $\forall i \in I_{b^+} \cup I_{b^-}, \#(p(i), b) > 1$.
(2) $\forall i \in I_{b^+}, \#(p(i), b^+) = \#(p(i), b)$.
(3) $\forall i \in I_{b^-}, \#(p(i), b^-) \geq 1$.
(4) $\forall i_1 \in I_{b^+}, \forall i_2 \in I_{b^-}, \#(p(i_1), b) = \#(p(i_2), b)$.
(5) $I_{b^+} \neq \emptyset \wedge I_{b^-} \neq \emptyset$.
(6) $\forall i_1, i_2 \in I_{b^+} \cup I_{b^-}, \forall b_1 \in b_h \cup b_l, b_1 \neq b,$
$\#(p(i_1), b_1) = \#(p(i_2), b_1) \wedge \#(p(i_1), b_1^+) = \#(p(i_2), b_1^+)$.

As with the SV pattern, we require that I_{b^+} and I_{b^-} are non-empty (5). We also require that all runs generated from inputs in I_{b^+} or I_{b^-} reach the branch b same number of times and agree on all branch condition evaluations other than the ones for the branch b (6). All inputs that cannot be put into I_{b^+} or I_{b^-} due to above constraints are put into I_0.

In the MVAF pattern the branch b is visited multiple times. I_{b^+} consists of input values that generate runs where at least one time when b is reached, the branch condition evaluates to true. I_{b^-} consists of input values that generate runs where every time b is reached, the branch condition evaluates to false. We formalize the MVAF pattern as:

(1) $\forall i \in I_{b^+} \cup I_{b^-}, \#(p(i), b) > 1$.
(2) $\forall i \in I_{b^+}, \#(p(i), b^+) \geq 1$.
(3) $\forall i \in I_{b^-}, \#(p(i), b^-) = \#(p(i), b)$.
(4) $\forall i_1 \in I_{b^+}, \forall i_2 \in I_{b^-}, \#(p(i_1), b) = \#(p(i_2), b)$.
(5) $I_{b^+} \neq \emptyset \wedge I_{b^-} \neq \emptyset$.
(6) $\forall i_1, i_2 \in I_{b^+} \cup I_{b^-}, \forall b_1 \in b_h \cup b_l, b_1 \neq b,$
$\#(p(i_1), b_1) = \#(p(i_2), b_1) \wedge \#(p(i_1), b_1^+) = \#(p(i_2), b_1^+)$.

We also define two more patterns by relaxing the requirements for the MVAT and MVAF patterns. For the MVATR pattern, I_{b^+} and I_{b^-} are defined as:

(1) $\forall i \in I_{b^+} \cup I_{b^-}, \#(p(i), b) \geq 0$.
(2) $\forall i \in I_{b^+}, \#(p(i), b^+) = \#(p(i), b)$.
(3) $\forall i \in I_{b^-}, \#(p(i), b^-) \geq 1$.
(4) $I_{b^+} \neq \emptyset \wedge I_{b^-} \neq \emptyset$.
(5) $\forall i_1, i_2 \in I_{b^+} \cup I_{b^-}, \forall b_1 \in b_h \cup b_l, b_1 \neq b,$
$\#(p(i_1), b_1) = \#(p(i_2), b_1) \Rightarrow \#(p(i_1), b_1^+) = \#(p(i_2), b_1^+)$.

where the requirement that branch condition b is reached the same number of times across runs generated by input in I_{b^+} or I_{b^-} is relaxed. This extends the applicability of the pattern to programs where the choice on condition b might cause the loop or recursive function containing b to terminate.

Similarly, we define the MVAFR pattern as:

(1) $\forall i \in I_{b^+} \cup I_{b^-}, \#(p(i), b) \geq 0$.
(2) $\forall i \in I_{b^+}, \#(p(i), b^+) \geq 1$.
(3) $\forall i \in I_{b^-}, \#(p(i), b^-) = \#(p(i), b)$.
(4) $I_{b^+} \neq \emptyset \wedge I_{b^-} \neq \emptyset$.
(5) $\forall i_1, i_2 \in I_{b^+} \cup I_{b^-}, \forall b_1 \in b_h \cup b_l, b_1 \neq b,$
$\#(p(i_1), b_1) = \#(p(i_2), b_1) \Rightarrow \#(p(i_1), b_1^+) = \#(p(i_2), b_1^+)$.

In the MVATR and MVAFR patterns, similar to the the SVR pattern, the requirement that all runs generated from input in I_{b^+} or I_{b^-} agree on all branch evaluations bar b itself is relaxed. We only require agreement on all branch conditions reached the same number of times across the runs. Once again, in practice we choose partitions that maximize the number of conditions agreed upon.

3.3 Branch Instrumentation

Given a program p we use dependency analysis discussed earlier to determine the set of secret dependent branches b_h. Then we instrument the program in order to generate sets of inputs based on the patterns above.

Given the program p and the secret dependent branches b_h we use Java bytecode manipulation and analysis framework ASM [14] to generate an instrumented program p_{inst} by introducing two counters (i.e., integer variables) for each secret-dependent branch b. One counter c_b is initialized to 0 and is incremented by one every time program execution reaches b. The other counter c_{b^+} is initialized to 0 and is incremented by one every time the program execution reaches b and the branch condition for b evaluates to true. The instrumented program p_{inst} prints the values of these counters when the program terminates. Given an input i, let $p_{inst}(i)[c_b]$ and $p_{inst}(i)[c_{b^+}]$ denote the values of c_b and c_{b^+} printed by p_{inst} when

it is run using input i. Then, we have the following equivalences:

$\#(p(i), b) = p_{inst}(i)[c_b]$, $\#(p(i), b^+) = p_{inst}(i)[c_{b^+}]$,

$\#(p(i), b^-) = p_{inst}(i)[c_b] - p_{inst}(i)[c_{b^+}]$.

Next, we generate a set of inputs $I_R \subseteq I$ and run p_{inst} on each $i \in I_R$. We focus on programs where the secret input is either string or numeric, and we use built-in Java functions returning pseudorandom values to build I_R. Then, we look at the output of each p_{inst} and using the patterns SV, MVAT, and MVAF and their relaxed variants we identify subsets $I_{b^+} \subseteq I_R$ and $I_{b^-} \subseteq I_R$ for each secret dependent branch b if the outputs generated by p_{inst} on I_R match these patterns. The result of this step is pairs of subsets $I_{b_1^+}, I_{b_1^-}; I_{b_2^+}, I_{b_2^-}; \ldots$, one per branch condition. We denote them as vectors $\vec{I}_{b^+}, \vec{I}_{b^-}$, and use them to search for JIT-induced side channels during the JVM Fuzzing phase.

3.4 Input Generation for Priming

JIT-induced side channels arise when a bias in the input distribution to a program causes a program path to "heat up," speeding up its execution relative to other program paths. To automate the detection of possible side channels arising from JIT, we generate a set of possible priming inputs $I_\pi \subseteq I$. The ideal set of priming inputs are values that exercise different program paths. During JVM Fuzzing phase we evaluate each priming input $i_\pi \in I_\pi$ to determine its effectiveness at inducing a timing side channel.

We use Kelinci [31], an interface to run the grey-box fuzzer American Fuzzy Lop (AFL) [53] on Java programs, to generate I_π. AFL is a genetic fuzzer designed to automatically discover interesting test cases that trigger new behaviors in the targeted program. AFL has been widely and successfully used, finding hundreds of high-impact vulnerabilities [54] and has a large community of active users. AFL is a coverage-based fuzzer, employing a variety of strategies and heuristics to generate inputs that traverse different paths in the program. It uses several different techniques such as bit-flipping and sequential insertion of known interesting integers to effectively trigger new program behaviors. AFL stores a witness $i \in I$ for each new program state found. We use the complete set of witnesses found upon termination of fuzzing as I_π. Though there is no guarantee that AFL will be able to generate an input for every program path, it has low overhead and less limitations in comparison to heavy-weight analyses such as symbolic execution, making it a better choice for our automated analysis.

4 JVM FUZZING FOR SIDE CHANNELS

Our goal is to determine if there is any priming input $i_\pi \in I_\pi$ such that priming in favor of i_π results in a timing side channel allowing an attacker to distinguish I_{b^+} from I_{b^-} for any two parition cells of the input domain as defined in Section 3.

Priming the JVM. We provide the pseudocode for our priming procedure in Algorithm 1. Given a certain priming amount n, a distribution α, a priming value i_π, and a set of other input values I_π - i_π, the program p is called n times in total. Of those n times, an α fraction of the calls are on the input i_π. The rest of the calls are on a randomly chosen input value drawn from I_π - i_π. Intuitively, the ratio α exists to model the case where the attacker does not have complete control over the JVM and therefore cannot prime

```
input : n (priming amount), α (ratio), i_π, I_π - i_π (priming inputs)
numItersBothSides ← 2(n − n · α);
numItersRemaining ← (n − numItersBothSides);
for i ← 1 to numItersBothSides do
    if i is odd then
        call p(i_π);
    else
        call p(random(I_π - i_π));
    end
end
for i ← 1 to numItersRemaining do
    call p(i_π);
end
```

Algorithm 1: Priming

perfectly in favor of i_π. The lower the α ratio is, the less control we assume the attacker has. We experiment with varied α values to explore what percentage of calls to p an attacker needs to control in order to induce a timing side channel. Similarly, we vary the value of n to explore the needed amount of priming. Both of these parameters inform under what scenarios an attacker will be able to induce a timing side channel in program p.

Evaluating JVM Vulnerability. For a given priming of the JVM, we wish to evaluate if a timing side channel has been introduced that leaks information about the membership of some secret input i to sets I_{b^+} and I_{b^-} based on the timing of $p(i)$. Algorithm 2 outlines this evaluation process. Given the subsets I_{b^+} and I_{b^-}, we first prime the JVM as described above and then time a call to p on a random input i drawn from I_{b^+}. We collect a timing distribution for I_{b^+} by repeating this process N times. Various sources of non-determinism, from system noise to variations in runtime decisions made by the JIT compiler, affect the time measurements. The higher the value of N, the more robust the statistical profile is to such noise. We repeat this process to also collect a timing distribution for I_{b^-}.

Given the timing distributions for I_{b^+} and I_{b^-}, we compute the conditional entropy between the membership of i and the timing of $p(i)$ after priming the JVM. From this, we report how much information about the membership of i we can expect to be leaked from a single time measurement. The membership of i encodes one bit of information. Therefore a value of 1 means full leakage of the membership of i and a value of 0 means no leakage.

Parameter Exploration for JVM Fuzzing. For each program p, we have a set of pairs of input cells $(\vec{I}_{b^+}, \vec{I}_{b^-})$ generated as described in Section 3. We also have a set of priming inputs I_π generated for the same program using Kelinci and AFL as described in Section 3. Now, we iterate over these parameters, in essence fuzzing the JVM in an attempt to find a JVM state vulnerable to a timing side channel. This process is described in Algorithm 3. For each $i_\pi \in I_\pi$ and pair of input sets $(I_{b^+}, I_{b^-}) \in (\vec{I}_{b^+}, \vec{I}_{b^-})$, we evaluate if priming in favor of i_π induces a side channel allowing us to learn the membership of an input i in sets I_{b^+} and I_{b^-}. The runtime decisions controlling JIT compilation are complex, meaning that the number of priming iterations and priming distribution can influence whether a timing channel is induced. Therefore, we also explore those parameters, iterating over a set of potential priming amounts and distributions.

```
input  : N (profiling amount), n (priming amount), α (ratio),
         i_more, I_less (priming inputs), I_b+, I_b- (profiling test input sets)
v_b+, v_b- ← two empty vectors to store timing profiles;
for i ← 1 to N do
    Prime(n, α, i_π, I_π - i_π );
    v_b+.append( Time(p(i_b+ ← random(I_b+)) // start with fresh JVM
end
for i ← 1 to N do
    Prime(n, α, i_π, I_π - i_π );
    v_b-.append( Time(p(i_b- ← random(I_b-)) // start with fresh JVM
end
ComputeConditionalEntropy(v_b+, v_b-);
```

Algorithm 2: Evaluation

```
input  : N (profiling amount), 𝒩 (priming amounts), A (ratios),
         I_π (priming inputs), (I⃗_b+, I⃗_b-) (profiling test input sets)
for (I_b+, I_b-) ∈ (I⃗_b+, I⃗_b-) do
    for i_π ∈ I_π do
        for n ∈ 𝒩 do
            for α ∈ A do
                VulnerabilityEvaluation(N, n, i_π, I_π - i_π, I_b+, I_b-)
            end
        end
    end
end
```

Algorithm 3: JVM Fuzzing

5 EXPERIMENTAL EVALUATION

Datasets. The Blazer dataset consists of 24 benchmarks drawn from a combination of challenge programs from the DARPA STAC program, classic examples from the literature [26, 33, 44], and microbenchmarks constructed by the Blazer authors. The 24 benchmarks consist of 12 unsafe programs and their "safe" variants. Table 3a shows the LOC for the safe Blazer variants (unsafe variants are similar. Four different program analysis tools for automated side-channel detection have used this benchmark for evaluation. Blazer [7], Themis [17], and CoCo-Channel[12] are state-of-the-art static analysis tools for detecting timing side channels in Java bytecode. They each report the vulnerability of the unsafe program variants and the safety of the patched versions. DiffFuzz [39] is a dynamic analysis tool for the detection of side channels. DiffFuzz catches an overflow bug resulting in a timing side channel in *loopAndbranch_safe* and considers that the safety of *gpt14_safe* is questionable depending on an observability threshold. Otherwise, DiffFuzz agrees with the verdicts of the other tools.

The Themis dataset consists of 10 unsafe programs and their "safe" variants. We evaluate on the 8 unsafe programs that contain timing side channels and their "safe" variants. Table 3b shows the LOC for the safe Themis variants. Outside of a benchmark from JDK6, the rest are Java programs collected from Github. Themis, CoCo-Channel and DiffFuzz have all used this benchmark for evaluation, agreeing on the safety of all safe variants, bar *jetty_safe* where DiffFuzz detects a subtle side channel.

The DiffFuzz dataset consists of 5 unsafe programs with timing side channels and their safe variants used in addition to the Blazer and Themis benchmarks to evaluate DiffFuzz [39]. Table 3c shows the LOC for the safe DiffFuzz variants. Four of these programs and their corresponding safe variants are drawn from the open source project Apache FtpServer [1] and the last is from an open source authentication plugin for Minecraft servers available on Github [2]. We exclude *ibasys* from the DiffFuzz dataset since its secret input is an image. Such complex data structures can be handled by our approach by providing pseudorandom input generators for such domains. However, as previously stated, our current implementation focuses on numeric and string secret domains.

Hardware Setup. All experiments were run on a computer equipped with an Intel i5-6600K CPU at 3.50 GHz and 32 GB of RAM running Ubuntu Linux 16.04 (Linux 4.4.0-103) and the Java 8 Platform Standard Edition, version 1.8.0_162, from OpenJDK.

Experimental Setup. Each program variant comes with a set of inputs marked as secret. For each safe program variant, we first conducted dependency analysis as described in Section 3.1, yielding a set of secret-dependent branches. Using these branches, we generated the instrumented programs per Section 3.3. We then proceeded to generate random secret input to the instrumented program. We used our pattern detection scheme described in Section 3.2 on the results to generate sets of possible partitions cells. Focusing on programs with numeric or string input values enabled us to write a pseudorandom input generator for each variant. For the modular exponentiation cases and password comparison functions, we placed an upper bound on the bit length of the input integer or length of the input string. We fixed any public input value arbitrarily. We began matching with the more strict patterns (SV, MVAT, MVAF), but if no partitions were found, we proceeded to their relaxed variants. We stopped the process for each applicable pattern as soon as a partition was found. The end result is a vector of sets $(\vec{I}_{b^+}, \vec{I}_{b^-})$.

We wrote a driver for Kelinci to generate the set of priming input values I_π for each safe variant. As AFL requires a directory of well-formed seed input, we generated one or two seed inputs per variant. We allowed Kelinci to run on each variant for at most one hour and used the set of witnesses for different program states found as I_π for the variant.

With the Input Generation Phase complete, we performed the JVM fuzzing given in Algorithm 3. We used a set of priming numbers, $n = \{1000, 10000, 100000, 500000\}$ and distributions, $\alpha = \{0.5, 0.7, 0.9, 0.95, 0.998, 0.999, 0.9999\}$ to determine the vulnerability of the resulting JVM primed in favor of each $i_\pi \in I_\pi$. We used a profiling amount of $N = 100$. As a baseline, we ran each safe variant with JIT disabled and gathered timing information for each $(\vec{I}_{b^+}, \vec{I}_{b^-})$ pair. Since priming the JVM does not matter in this case, we did not explore the parameter space for these runs.

To determine the strength of the resulting timing channel, we calculate the entropy of partition cell membership conditioned on timing observation. The leakage reported is 1− this conditional entropy. We profile multiple times to gain probability distributions for observed execution times.

As noted by the authors of DiffFuzz [39], a NullPointerException in *unixlogin_safe* prevents it from being executable. We fixed the issue by replacing the culprit hash comparison with a dummy comparison. As discussed by the authors of CoCo-Channel, some secret-dependent conditionals (such as certain null pointer checks) in the Themis dataset were manually marked by the Themis authors as secret-independent. We ignored the same set of secret-dependent branches in order to obtain comparable results.

Table 1: Experimental results for Blazer (top), Themis (middle) and DiffFuzz (bottom) safe variants.

Program variant	Inst. pattern	Best leakage						Limited leakage						Leakage w/o JIT			
		leakage	n	α	avg. time (ns)			leakage	n	α	avg. time (ns)			leakage	avg. time (ns)		
					I_{b^+}	I_{b^-}	diff				I_{b^+}	I_{b^-}	diff		I_{b^+}	I_{b^-}	diff
array_safe	SV	0.995	100000	0.9990	205	16335	16130	0.741	1000	0.9000	359	434	75	0.029	286	287	1
loopBranch_safe	SV	1.000	100000	0.9990	192	16020	15828	0.953	1000	0.9000	476	338	138	0.206	267	287	20
sanity_safe	SV	0.995	100000	0.9990	199	13382	13183	0.337	1000	0.9000	725	685	40	0.026	343	350	7
straightline_safe	SV	0.995	100000	0.9980	200	15927	15727	0.810	1000	0.9000	397	477	80	0.025	373	377	4
unixlogin_safe	MVAF	0.995	10000	0.9980	329	17553	17224	0.519	1000	0.9000	620	639	19	0.114	604	640	36
modPow1_safe	MVAT	0.977	1000	0.9000	16977	8512	8465	0.977	1000	0.9000	16977	8512	8465	0.033	37095	37342	247
modPow2_safe	MVAT	0.995	1000	0.9990	34423	7853	26570	0.971	1000	0.9000	24520	8809	15711	0.397	36491	38220	1729
pwdEqual_safe	MVAF	0.485	500000	0.9999	547	12870	12323	0.357	10000	0.7000	378	344	34	0.077	857	822	35
pwdEqual_safe	MVAT	0.744	500000	0.9999	550	15877	15337	0.597	10000	0.9000	400	294	106	0.363	782	881	99
pwdEqual_safe	SV	0.849	500000	0.9990	18642	529	18113	0.635	1000	0.9000	723	930	207	0.051	877	902	25
k96_safe	MVAT	0.995	1000	0.9000	19952	8547	11405	0.995	1000	0.9000	19952	8547	11405	0.078	37065	37276	211
gpt14_safe	MVAT	0.862	10000	0.9980	15434	43265	27831	0.306	10000	0.9000	28290	21738	6552	0.071	37305	37654	349
login_safe	SV	0.967	500000	0.9999	30769	3191	27578	0.571	10000	0.7000	432	556	124	0.354	3683902128	3686425340	2523212
jetty_safe	SV	1.000	100000	0.9980	302	16617	16315	0.510	1000	0.9000	610	752	142	0.120	720	759	39
jetty_safe	MVAF	0.873	100000	0.9980	891	16108	15217	0.253	1000	0.9000	624	687	63	0.026	722	726	4
jetty_safe	MVAT	0.860	100000	0.9990	271	16842	16571	0.178	10000	0.9000	291	362	71	0.074	721	751	30
spring_safe	SV	1.000	100000	0.9980	719	19123	18404	0.067	10000	0.7000	697	774	77	0.015	4111	4198	87
tomcat_safe	SVR	1.000	100000	0.9990	8943	261209	252266	1.000	1000	0.9000	16801	184082	167281	1.000	27907	168234	140327
pac4j_safe	SVR	1.000	10000	0.9980	19223	188121	168898	1.000	10000	0.7000	20076	95022	74946	0.980	270478	335170	64692
apache_clear_safe	SV	0.970	10000	0.9990	4677	50015	45338	0.026	1000	0.7000	5695	5721	26	0.020	195067	196100	1033
apache_stringUtils_safe	MVAF	0.612	1000	0.9500	1612	2014	402	0.307	10000	0.9000	1116	1375	259	0.352	5391	5709	318

Table 2: Experimental results for Blazer (top), Themis (middle) and DiffFuzz (bottom) unsafe variants.

Program variant	Inst. pattern	Best leakage						Worst leakage					
		leakage	n	α	avg. time (ns)			leakage	n	α	avg. time (ns)		
					I_{b^+}	I_{b^-}	diff				I_{b^+}	I_{b^-}	diff
array_unsafe	SV	0.995	1000	0.9000	345	234	111	0.148	100000	0.5000	255	240	15
loopBranch_unsafe	SV	0.924	100000	0.9990	16325	197	16128	0.457	10000	0.9000	707	253	454
notaint_unsafe	SV	1.000	1000	0.5000	1184025	306	1183719	0.868	100000	0.9990	244693159	16331	244676828
sanity_unsafe	SV	0.995	100000	0.9980	13757	204	13553	0.111	10000	0.9990	5987	233	5754
straightline_unsafe	SV	0.995	100000	0.9990	16136	196	15940	0.148	100000	0.7000	201	199	2
unixlogin_unsafe	MVAF	0.459	1000	0.7000	544	598	54	0.160	10000	0.5000	213	223	10
modPow1_unsafe	MVAT	0.287	100000	0.9500	5661	4716	945	0.104	1000	0.5000	3250	3300	50
modPow2_unsafe	MVAT	0.256	1000	0.9000	2329	2802	473	0.126	100000	0.5000	20512	20595	83
pwdEqual_unsafe	MVAF	0.353	100000	0.5000	614	574	40	0.185	1000	0.5000	562	520	42
pwdEqual_unsafe	MVAT	0.813	1000	0.9000	809	652	157	0.189	100000	0.9000	419	421	2
pwdEqual_unsafe	SV	0.664	1000	0.9980	691	573	118	0.202	10000	0.9500	375	366	9
k96_unsafe	MVAT	0.821	10000	0.9990	76023	28853	47170	0.176	10000	0.7000	11460	10886	574
gpt14_unsafe	MVAT	0.995	100000	0.9990	1554	35114	33560	0.099	10000	0.7000	4233	4641	408
login_unsafe	SV	0.937	500000	0.9999	5116	29735	24619	0.443	1000	0.5000	680	526	154
bootauth_unsafe	MVATR	0.980	100000	0.9980	3006	30653	27647	0.022	1000	0.5000	14085	14169	84
bootauth_unsafe	MVAFR	1.000	100000	0.9980	3049	39056	36007	0.044	1000	0.5000	12084	12282	198
jdk_unsafe	MVATR	0.710	100000	0.9980	1073	18290	17217	0.006	1000	0.5000	3173	3963	790
jdk_unsafe	MVAFR	0.400	100000	0.9980	2272	14573	12301	0.007	1000	0.5000	1019	1032	13
jetty_unsafe	SV	1.000	10000	0.9000	307	749	442	0.740	500000	0.9000	457	894	437
jetty_unsafe	MVAFR	0.554	100000	0.9990	369	15151	14782	0.033	1000	0.5000	765	771	6
jetty_unsafe	MVATR	0.780	1000	0.9990	510	1019	509	0.002	10000	0.5000	306	322	16
orientdb_unsafe	MVATR	1.000	500000	0.9999	436	9818	9382	0.024	1000	0.5000	531	559	28
orientdb_unsafe	MVAFR	0.244	1000	0.9500	623	735	112	0.038	10000	0.5000	363	395	32
picketbox_unsafe	MVATR	0.755	100000	0.9990	795	19197	18402	0.013	1000	0.5000	448	487	39
picketbox_unsafe	MVAFR	0.417	500000	0.9999	1188	6925	5737	0.063	1000	0.5000	616	641	25
spring_unsafe	SV	1.000	100000	0.9990	656	19705	19049	0.012	1000	0.5000	2479	2563	84
tomcat_unsafe	SVR	1.000	10000	0.7000	6699	246234	239535	0.810	1000	0.5000	17496	208251	190755
pac4j_unsafe	SVR	1.000	10000	0.9000	18232	106287	88055	0.498	10000	0.5000	20891	32312	11421
apache_clear_unsafe	SV	1.000	100000	0.9990	311	19244	18933	0.181	10000	0.5000	329	426	97
apache_md5_unsafe	MVAFR	0.016	1000	0.9990	15704	16138	434	0.009	1000	0.5000	14082	14197	115
apache_stringUtils_unsafe	MVAF	1.000	1000	0.9990	461	2732	2271	0.341	10000	0.5000	505	920	415
authMeReloaded_unsafe	MVAFR	0.025	10000	0.9990	197046	198718	1672	0.003	1000	0.9000	203260	207381	4121

Table 3: Size of experimental subjects

(a) Dataset information for Blazer

Program variant	Program size (LOC)
array_safe	17
loopBranch_safe	27
nosecret_safe	5
sanity_safe	18
straightline_safe	13
unixlogin_safe	53
modPow1_safe	14
modPow2_safe	17
pwdEqual_safe	18
k96_safe	27
gpt14_safe	12
login_safe	? 21

(b) Dataset information for Themis

Program variant	Program size (LOC)
bootauth_safe	74
jdk_safe	36
jetty_safe	77
orientdb_safe	134
picketbox_safe	208
spring_safe	41
tomcat_safe	100
pac4j_safe	104

(c) Dataset information for DiffFuzz

Program variant	Program size (LOC)
apache_clear_safe	186
apache_md5_safe	159
apache_saltedPW_safe	191
apache_stringUtils_safe	29
authmreloaded_safe	40 ?

We also evaluated our approach on the unsafe program variants in order to explore how JIT can impact timing side channels already present at the source code level. In most cases, the secret-dependent branches of the safe variants can be matched to corresponding branchs in their unsafe variants. Hence, we evaluate the separability of the same $(\vec{I}_{b^+}, \vec{I}_{b^-})$ pairs across the safe and unsafe variants. This allows us to evaluate the appearance of the "same" timing side channel across both versions. Likewise, we used the same priming input across matching variants. We note that the public values chosen are not necessarily those which maximize the existing side channel in the unsafe variants. In fact, the $(\vec{I}_{b^+}, \vec{I}_{b^-})$ pairs do not even necessarily correspond to the timing channels originally present in the unsafe variant. The experiments on the unsafe versions are not aimed at finding the strongest side channels, but are shown as a comparison for the behavior found in the safe variants. The *nosecret_safe*, *bootauth_safe*, *jdk_safe*, *orientdb_safe*, *picketbox_safe* and *authMeReloaded_safe* variants have no secret-dependent branches and the *apache_md5_safe* and *apache_salted_safe* variants have no secret-dependent branches where both evaluations of the branch are possible. Therefore, we performed our instrumentation and fuzzing technique on their unsafe variants to generate $(\vec{I}_{b^+}, \vec{I}_{b^-})$ and I_π in order to evaluate those programs.

5.1 Experimental Results and Discussion

The results of our experiments on the "safe" Blazer, Themis and DiffFuzz variants are given in Table 1. We report the name of the variant under test and the partition template matched. We report the highest leakage we observed, the parameters (priming number and distribution) leading to that leakage, and the average execution times of the program on input values from the sets I_{b^+} and I_{b^-} and the difference between these average execution times. We report the average execution times to give a sense of the observability of the resulting side channel. This is a consideration for an attacker with more limited timing capabilities and provides a more refined characterization of the side channel. Blazer, Themis, CoCo-Channel and DiffFuzz all report a static variant of the maximal cost difference across program paths; what we report is the runtime counterpart, which is a more realistic measure for observability.

We additionally report two other scenarios. First, we report the leakage and difference in execution time when JIT is disabled. These results provide a base line and are used to evaluate to what degree JIT optimization is responsible for any timing side channels as opposed to secret-dependent timing imbalance in the source code. Then, we report the maximum leakage and difference in execution time under what we call the "limited" priming scenario. In this scenario, the number of priming iterations is capped at 10,000 and the strongest distribution considered is 0.9. This scenario gives us a sense of how well an attacker with limited capabilities could leverage JIT to induce side channels.

Our automated framework was able to induce large timing side channels in all "safe" program variants in the Blazer dataset with the exception of *notaint_safe*. These results contradict the results reported by four state-of-the-art analysis tools. We were able to induce large timing side channels in the Themis variants *jetty_safe* and *spring_safe* and the DiffFuzz variants *apache_clear_safe* and *apache_stringUtils_safe*. We also observed timing side channels in *tomcat_safe* and *pac4j_safe*, though we found these side channels to be present (though mitigated) even when JIT is disabled. The remaining safe Themis variants along with DiffFuzz *authMeReloaded_safe* and Blazer variant *notaint_safe* do not contain any secret dependent branches. The DiffFuzz variants *apache_md5_safe* and *apache_saltedPW_safe* contain a secret-dependent branch but only one evaluation of that branch is satisfiable.

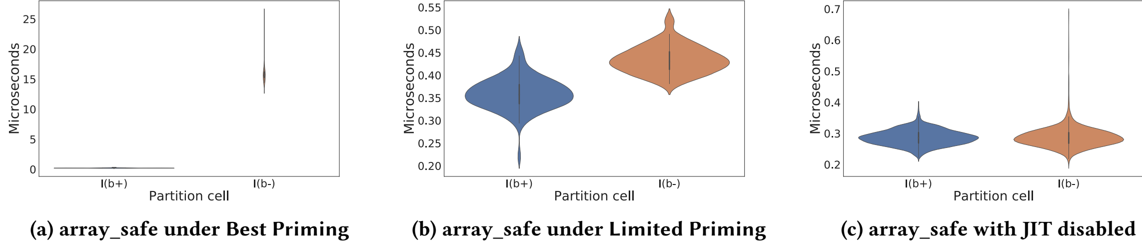
Our results are validated at runtime, demonstrating the existence of the timing channels. In many cases, the conditional entropy reported is close to 1, meaning that an attacker would be able to deduce the membership of the secret input with almost certainty after one timing observation. Additionally, the difference in the average execution time of input from $I_{b^+}$and those from I_{b^-} can be on the order of tens of microseconds even for Blazer variants where the largest safe variant has only 20 basic blocks.

There are two scenarios in which no side channel was found in the safe variants: 1) when there is no secret dependent branch and 2) when any secret-dependent branch always evaluates to the same result. For password checking functions, safe variants were created by replacing secret-dependent branches with bit-wise manipulation code and, hence, these safe variants are resilient to JIT-induced side channels. Our observations point to potential mitigation techniques against JIT-induced side channels.

Our results for the case when JIT is disabled confirm that JIT compilation is responsible for the timing side channels. In almost all cases, disabling JIT eliminates the timing side channel and the information leakage. In *loopBranch_safe*, a small timing channel is reported. This is consistent with source-code-level analysis which reports a small imbalance between the relevant program paths. The small magnitude of this side channel implies that it is not likely to be observable in scenarios where our much larger JIT-induced timing channels would be. A timing side channel is also present in *modPow2_safe*. We believe this is because of slightly different multiplication done across the relevant paths which may vary in cost. Similar reasoning explains the timing side channel in *apache_stringUtils_safe*, where a variable cost instruction is likely the cause of the smaller side channel observed. We believe the side channel in *pwdEqual_safe* (MVAT) might be due to CPU level branch prediction. Finally, the small side channel reported in *login_safe* is likely due to the small number of samples taken (100) relative to the noise of the program.

As mentioned earlier, *tomcat_safe* and *pac4j_safe* both contain side channels even when JIT is disabled. In the case of *tomcat_safe*, this is congruous with the observation made by both the Themis and CoCo-Channel authors that the side channel is mitigated in the safe variant but not removed. Significant non-trivial computation is performed in this application and it is possible that the static models used in previous tools did not capture the program cost realistically. Likewise, *tomcat_safe* involves error handling code concerning interactions with an external database and it is likely that the static models for these computations were approximate.

Our results in the Limited Priming Scenario are also informative. In all cases except *apache_string_safe*, the amount of leakage about the membership of the secret input is higher than the scenario when JIT is disabled. In some cases, such as *jetty_safe*, the resulting timing side channel is significantly smaller than the Best Priming Scenario and only tens of nanoseconds separate the timings of input



(a) array_safe under Best Priming **(b) array_safe under Limited Priming** **(c) array_safe with JIT disabled**

Figure 4: Execution time distributions for the array_safe variant under the Best Priming, Limited Priming and without JIT.

from the partition cells. In other cases such as *k96_safe*, however, a timing side channel of tens of thousands of nanoseconds separates the timing distributions. Thus, even an attacker with more limited capabilities may be able to leverage JIT-induced side channels.

The results of our experiments on the unsafe Blazer, Themis and DiffFuzz variants are given in Table 2. We do not report results for the *apache_saltedPW_unsafe* variant of DiffFuzz since we do not find any input matching one of our partition patterns. Again, we do not aim to find the strongest side channel in the unsafe variant but to evaluate the impact of different priming strategies on the behavior of side channels. For example, our results on *bootauth_unsafe* show that a powerful side channel present in some JVM states all but disappears in others. In some cases, most notably *modpow1_unsafe* and *modpow2_unsafe*, no extremely strong side channel is detected. As mentioned previously, the public input values used in these experiments are not guaranteed to be values for which the intended side channel is strong in the unsafe variants. Additionally, the programs do differ from their safe variants so a priming strategy successful in inducing a timing side channel in the safe version might not be in the unsafe one. We believe this explains the lower leakage for some programs. Overall, our results on the unsafe variants further demonstrate the significant impact the runtime can have over timing side channels.

Coverage of Secret-Dependent Branches. In a few safe variants, additional branches were marked as secret dependent. In both *loopBranch_safe* and *login_safe*, an additional branch was marked but, as in the *apache_md5_safe* and *apache_saltedPW_safe* variants, the condition associated with the branch could only ever be evaluated one way. Any path with the opposite evaluation is unrealizable. Therefore, there are no results reported for these cases. *k96_safe*, *gpt14_safe*, *modpow1_safe* *modpow2_safe* and *apache_stringUtils_safe* all have a branch condition nested within a loop but we only report results for either the MVAT or MVAF pattern. The other pattern is not applicable as the branch condition must evaluate to true (or false in the case of *apache_stringUtils_safe*) at least once. *unixlogin_safe* also marks a branch condition inside a loop as secret dependent, but only the MVAF pattern is evaluated. Neither the input generation responsible for (I_{b^+}, I_{b^-}) or AFL could find input satisfying the MVAF pattern. In this case, such an input exists but is very specific. Running AFL for longer or providing better seed input might enable it to find the input. Evaluation of *apache_md5_unsafe* and *authMeReloaded_unsafe* is limited to MVAFR for the same reason.

Importance of Priming Input. Table 1 reports the leakage values for the best priming input i_π per variant. A more nuanced look at the experiments is needed to understand the impact of the choice of priming value. In some cases, such as *straightline_safe*, all priming input values generated by Kelinci were effective at introducing a timing side channel. This simple program has only one secret-dependent branch, and the timing side channel is introduced regardless of which of the two program paths is favored. In other cases, such as *pwdEqual_safe* (MVAT), only a single priming input effectively introduced a timing side channel.

Relation of (I_{b^+}, I_{b^-}) to the secret input. How much danger an attacker being able to separate (I_{b^+}, I_{b^-}) poses can be answered in part by considering the size of the (I_{b^+}, I_{b^-}) relative to the entire input domain. In some programs, such as *straightline_safe* and *array_safe*, $I_{b^+} \cup I_{b^-}$ is the entire secret domain for the fixed public input. In others, such as the MVAF pattern of *jetty_safe*, (I_{b^+}, I_{b^-}) is the entire domain of possible secret input of a fixed length (we chose length 4). This means that if an attacker is aware that the secret input is of length 4, she will be able to determine the membership of that input. Another part of the answer depends on the relative sizes of (I_{b^+}, I_{b^-}) to each other. In *sanity_safe*, for instance, the set of possible secret inputs is evenly divided between ($I_{b^+}$and I_{b^-}). However, in the MVAT pattern of *jetty_safe*, the only string of length 4 always matching a public string of length 4 is that public string itself. Therefore, answering the membership query is equivalent to determining whether or not the secret is a certain exact string. There are certainly applications where even this is dangerous (suppose a branch condition allows you to learn if an anonymous user is in fact a particular celebrity); however, less information about the secret itself can be learned in this case. Ultimately, we demonstrate the *existence* of *a* side channel. This is the same question that Blazer, Themis, CoCo-Channel and DiffFuzz concern themselves with. The follow up question of how much information the side channel leaks about the secret itself is left to other analyses.

Potential Patches. Our results indicate that any JIT-enabled system is vulnerable to a timing side channel arising from biased input distributions. One potential patch is to disable JIT. However, doing so critically impacts the performance of the JVM. For example, the execution time of one call to *login_safe* is under 500 nanoseconds with JIT optimization and *over 3 seconds* when JIT is disabled. Disabling JIT slows this program by a factor of more than 7000000 times. Such performance loss would render Java programs unusable for many applications. Selectively disabling JIT only for highly sensitive, security-critical code would mitigate this performance loss but ultimately results in a trade-of between performance and security that needs to be reconciled. Another potential option would be

replace just-in-time compilation with ahead-of-time compilation (AoT) [46], where all optimization and code generation is done *before* running the program and runtime statistics are not used in optimization decisions. It is thus likely to be more robust against side channels introduced at runtime and this additional security could be another reason to adopt it for certain security-critical methods. Finally, our results on variants such as *notaint_safe* motivates replacing secret dependent branches with bit-wise manipulation code as a mitigation.

6 RELATED WORK

JIT-Induced Side Channels. The fact that dynamic compilation can introduce side-channel vulnerabilities has been noted before [42]. More recently, JIT compilation has been shown to be responsible for side-channel vulnerabilities in open source Java applications that leak information even over the noisy public internet [11]. Our work addresses a hitherto unexplored dimension of JIT-induced side channels – the automation of their detection. We present a framework for systematically exploring the huge set of possible JVM states and evaluate their vulnerability to side channels. Our focus on automation, and the framework we present for automation, clearly differentiates our contributions from earlier work on this class of side channels [11, 42]. We also evaluate our technique on widely-used datasets from the literature, providing the first evaluation of this new class of side channels on existent datasets.

Compiler-based Mitigation Strategies. Compiler-based mitigation techniques for side-channel vulnerabilities have been considered [20, 21, 49]. In fact, leveraging the statistical profiling information which is the crux of JIT-induced side channels has been proposed as part of a mitigation strategy [49]. We believe that integrating these strategies into the JVM has potential as a possible remedy for JIT-induced side channels, though their effectiveness would require investigation. However, at present none of these techniques have been integrated with the most widely used JVM, HotSpot, which remains vulnerable to JIT-induced side channels.

Side-Channel Detection. A significant body of research has been done in the field of software side-channel detection. Blazer [7] performs a grammar-based analysis to decompose execution traces into different partitions and then verifies properties of each partition to prove a program's resilience to timing side channels. Themis introduces a variant of Cartesian Hoare Logic concerned with a program's resource usage to generate proofs about the maximal cost difference across program paths. CoCo-Channel [12] performs a compositional analysis, generating a symbolic cost expression for each program component and using SMT solvers to answer pertinent questions about the maximal cost difference incurred by components. All three of these static analysis tools are ignorant of any potential impact the runtime might have over a program's execution time. They only consider the source code of a program under test and, as such, they are unable to detect JIT-induced side channels. DiffFuzz [39] is a differential fuzzing technique aimed towards finding program executions that maximize a cost difference. This dynamic tool confirms the safety of the safe Blazer variants, because, like its static counterparts, it does not consider the JVM's state during its exploration. Differential analyses are also used to automatically detect vulnerabilities in SSL/TLS [50]. Other dynamic approaches to side-channel detection analyze the network traffic of web applications in a black-box manner [16, 37, 47]. Type-based approaches to side-channel detection [6, 28] and approaches specific to cache-side channels [22, 27] have also been proposed. Currently, no static or dynamic approach to side-channel detection considers the state of the JVM as a factor in assesing side channels.

Other research is aimed towards the quantification of side-channel vulnerabilities [9, 43]. Research in this area aims at asking not simply if a program leaks information through a side channel, but *how much* information about the secret is leaked. These analyses rely on symbolic execution and suffer from poor scalability. Extensions to these approaches to perform attack synthesis on programs vulnerable to side channels [10, 45, 48] have also been explored.

CPU-induced side-channels. Other classes of side-channel attacks such as cache attacks and branch prediction analysis (BPA) attacks leverage the runtime-dependent behavior of CPUs. Cache-based side-channel attacks [13, 30, 36, 41, 51] have been theoretically analyzed for years and have also been demonstrated as very powerful techniques to recover sensitive information in pragmatic scenarios. Side channels can also be introduced into security-sensitive code through the branch prediction mechanism of the CPU [3–5]. As a result, the CPU's branch predictor unit has been exploited in different flavours of timing side-channels [23, 24, 34]. All this work considers the impact of runtime behavior on the processor. We instead concern ourselves with its impact on the JVM.

JVM testing. There is prior work on testing JVM implementations [15, 18, 19, 52]. Our work does not look for errors in JVM implementations, but rather investigates if and when JIT compilation causes timing side channels.

7 CONCLUSIONS

We demonstrated that timing side channels in software are not a static phenomenon but can be introduced dynamically into programs through biased input distributions. In particular, the just-in-time (JIT) compilation mechanism, which is critical for the performance of Java programs, can be leveraged to introduce timing channels into programs that do not contain side-channel vulnerabilities when JIT is disabled. We have developed and evaluated an automatic technique to fuzz the JVM in order to detect JIT-induced side channels. We use this technique to show that the previously safe labeled variants of the well-known and studied benchmarks are vulnerable to JIT-induced timing side channels, contradicting the results of four state-of-the-art analysis tools. We conclude that JIT-induced side channels are prevalent and the side channel vulnerability detection techniques have to take into account the impact of a program's runtime during side channels analysis.

REFERENCES

[1] [n. d.]. Apache FtpServer. https://mina.apache.org/ftpserver-project/. Accessed: 2018-08-21. ([n. d.]).

[2] [n. d.]. Authentication plugin for the Bukkit/Spigot API. https://github.com/AuthMe/AuthMeReloaded. Accessed: 2018-08-21. ([n. d.]).

[3] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. 2007. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In *IMA International Conference on Cryptography and Coding*. Springer, 185–203.

[4] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security.* ACM, 312–320.

[5] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. Predicting secret keys via branch prediction. In *Cryptographers Track at the RSA Conference*. Springer, 225–242.

[6] Johan Agat. 2000. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 40–53.

[7] 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 *ACM SIGPLAN Notices*, Vol. 52. ACM, 362–375.

[8] Daniel Balasubramanian, Zhenkai Zhang, Dan McDermet, and Gabor Karsai. 2017. Janalyzer: A Static Analysis Tool for Java Bytecode. *ISIS* 17 (2017), 104.

[9] 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*. ACM, 193–204.

[10] L. Bang, N. Rosner, and T. Bultan. 2018. Online Synthesis of Adaptive Side-Channel Attacks Based On Noisy Observations. In *2018 IEEE European Symposium on Security and Privacy (EuroS P)*. 307–322. https://doi.org/10.1109/EuroSP.2018.00029

[11] Tegan Brennan, Nicolás Rosner, and Tevfik Bultan. 2020. JIT Leaks: Inducing Timing Side Channels through Just-In-Time Compilation. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (to appear)*.

[12] Tegan Brennan, Seemanta Saha, Tevfik Bultan, and Corina S Păsăreanu. 2018. Symbolic path cost analysis for side-channel detection. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 27–37.

[13] Billy Bob Brumley and Risto M Hakala. 2009. Cache-timing template attacks. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 667–684.

[14] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*.

[15] Andrea Calvagna and Emiliano Tramontana. 2013. Automated Conformance Testing of Java Virtual Machines. In *Seventh International Conference on Complex, Intelligent, and Software Intensive Systems, CISIS 2013, Taichung, Taiwan, July 3-5, 2013*. 547–552.

[16] Peter Chapman and David Evans. 2011. Automated black-box detection of side-channel vulnerabilities in web applications. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 263–274.

[17] 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*. ACM, 875–890.

[18] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 1257–1268.

[19] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 85–99.

[20] Jeroen V Cleemput, Bart Coppens, and Bjorn De Sutter. 2012. Compiler mitigations for time attacks on modern x86 processors. *ACM Transactions on Architecture and Code Optimization (TACO)* 8, 4 (2012), 23.

[21] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. 2009. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 45–60.

[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 (TISSEC)* 18, 1 (2015), 4.

[23] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–13.

[24] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 693–707.

[25] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24041

[26] Daniel Genkin, Itamar Pipman, and Eran Tromer. 2015. Get your hands off my laptop: Physical side-channel key-extraction attacks on PCs. *Journal of Cryptographic Engineering* 5, 2 (2015), 95–112.

[27] Shengjian Guo, Meng Wu, and Chao Wang. 2018. Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 377–388.

[28] Daniel Hedin and David Sands. 2005. Timing aware information flow security for a javacard-like bytecode. *Electronic Notes in Theoretical Computer Science* 141, 1 (2005), 163–182.

[29] S. Horwitz, T. Reps, and D. Binkley. 1988. Interprocedural Slicing Using Dependence Graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*. ACM, New York, NY, USA, 35–46. https://doi.org/10.1145/53990.53994

[30] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. 1998. Side channel cryptanalysis of product ciphers. In *European Symposium on Research in Computer Security*. Springer, 97–110.

[31] Rody Kersten, Kasper Luckow, and Corina S Păsăreanu. 2017. POSTER: AFL-based Fuzzing for Java with Kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2511–2513.

[32] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203* (2018).

[33] Paul C Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*. Springer, 104–113.

[34] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium, USENIX Security*. 16–18.

[35] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

[36] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 605–622.

[37] Luke Mather and Elisabeth Oswald. 2012. Quantifying Side-Channel Information Leakage from Web Applications. *IACR Cryptology ePrint Archive* 2012 (2012), 269.

[38] Flemming Nielson, Hanne R Nielson, and Chris Hankin. 2015. *Principles of program analysis*. Springer.

[39] Shirin Nilizadeh, Yannic Noller, and Corina S Pasareanu. 2018. DifFuzz: Differential Fuzzing for Side-Channel Analysis. *arXiv preprint arXiv:1811.07005* (2018).

[40] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers' Track at the RSA Conference*. Springer, 1–20.

[41] Dan Page. 2002. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive* 2002 (2002), 169.

[42] Daniel Page. 2006. A Note On Side Channels Resulting From Dynamic Compilation. https://eprint.iacr.org/2006/349.pdf. *Cryptology ePrint archive* (2006).

[43] 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 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 387–400. https://doi.org/10.1109/CSF.2016.34

[44] Corina S Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. 2016. Multi-run side-channel analysis using Symbolic Execution and Max-SMT. In *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*. IEEE, 387–400.

[45] Quoc-Sang Phan, Lucas Bang, Corina S Pasareanu, Pasquale Malacaria, and Tevfik Bultan. 2017. Synthesis of Adaptive Side-Channel Attacks. In *Computer Security Foundations Symposium (CSF), 2017 IEEE 30th*. IEEE, 328–342.

[46] Todd A Proebsting, Gregg M Townsend, Patrick G Bridges, John H Hartman, Tim Newsham, and Scott A Watterson. 1997. Toba: Java for Applications-A Way Ahead of Time (WAT) Compiler.. In *COOTS*. 41–54.

[47] Nicolás Rosner, Ismet Burak Kadron, Lucas Bang, and Tevfik Bultan. 2018. *Profit: Detecting and Quantifying Side Channels in Networked Applications*.

[48] Seemanta Saha, Ismet Burak Kadron, William Eiers, Lucas Bang, and Tevfik Bultan. 2019. Attack Synthesis for Strings Using Meta-Heuristics. *SIGSOFT Softw. Eng. Notes* 43, 4 (Jan. 2019), 56–56. https://doi.org/10.1145/3282517.3282527

[49] Jeroen Van Cleemput, Bjorn De Sutter, and Koen De Bosschere. 2017. Adaptive compiler strategies for mitigating timing side channel attacks. *IEEE Transactions on Dependable and Secure Computing* (2017).

[50] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. 2017. STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 859–874. https://doi.org/10.1145/3133956.3134016

[51] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.. In *USENIX Security Symposium*. 719–732.

[52] Takahide Yoshikawa, Kouya Shimura, and Toshihiro Ozawa. 2003. Random Program Generator for Java JIT Compiler Test System. In *3rd International Conference on Quality Software (QSIC 2003), 6-7 November 2003, Dallas, TX, USA*. 20.

[53] Michal Zalewski. 2017. American fuzzy lop (AFL) fuzzer. *http://lcamtuf.coredump.cx/afl/* (2017).

[54] Michal Zalewski. 2017. The bug-o-rama trophy case. *http://lcamtuf.coredump.cx/afl/#bugs* (2017).

[55] Kehuan Zhang, Zhou Li, Rui Wang, XiaoFeng Wang, and Shuo Chen. 2010. Sidebuster: automated detection and quantification of side-channel leaks in web application development. In *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 595–606.