Inducing JIT-Based Side Channels for Inferring Predicates about Secrets

Tegan Brennan, Nicolás Rosner and Tevfik Bultan
University of California Santa Barbara
{tegan, rosner, bultan}@cs.ucsb.edu

***Abstract*—Side-channel vulnerabilities in software are caused by an observable imbalance in resource usage across different program paths. In this paper we demonstrate that just-in-time (JIT) compilation, which is crucial to the runtime performance of modern Java virtual machines (JVMs), can be leveraged to induce timing side channels. We present a technique for creating dynamic, JIT-based side channels and using them to learn values of predicates about secret inputs. Our technique includes a mechanism for priming the state of the JVM to trigger certain runtime JIT optimizations. The timing of subsequent method calls then leaks information about the values of predicates (branch conditions) on inputs. We define two attack models and five vulnerability templates based on the optimizations performed by state-of-the-art JIT compilers. Applying these templates to Java methods gives rise to various types of runtime-behavior-dependent side-channel vulnerabilities. We use symbolic execution to automatically generate input values that prime the JVM into states that foster such vulnerabilities. We evaluate our technique on three widely used classes from the Java standard library: java.lang.Math, java.lang.String, and java.math.BigInteger. We show 18 attempts to induce JIT-based vulnerabilities, including successful and unsuccessful ones, and discuss the results in detail. The significant amount of information leakage achieved demonstrates the viability and potential of this class of attacks.**

***Index Terms*—program analysis, side channel analysis, just-in-time compilation.**

I. Introduction

Cyber-attacks stealing confidential information are becoming increasingly frequent and devastating as modern software systems store and manipulate greater amounts of sensitive data. Leaking information about private user data, such as the financial and medical records of individuals, trade secrets of companies and military secrets of states can have drastic consequences. Although programs that have access to secret information are expected to protect it, many software systems contain vulnerabilities that leak information.

By observing non-functional side effects of software systems such as execution time or memory usage, *side-channel* attacks can capture secret information. Though side-channel vulnerabilities have been known for decades [1], they are still often neglected by software developers. They are commonly thought of as impractical despite a growing number of demonstrations of realistic side-channel attacks that result in critical security vulnerabilities [2]–[4]. For instance, exploitable timing channel information flows were discovered for Google's Keyczar Library [5], the Xbox 360 [6], implementations of RSA encryption [2], and the open authorization protocol OAuth [7]. These vulnerabilities highlight the need for preemptive discovery of side channel vulnerabilities and their removal from software.

This material is based on research sponsored by 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.

```
public bool check(String guess) {
   for(int i=0; i<guess.len; i++) {
      if (guess[i] != password[i])
         return false;
   }
   return true;
}
```

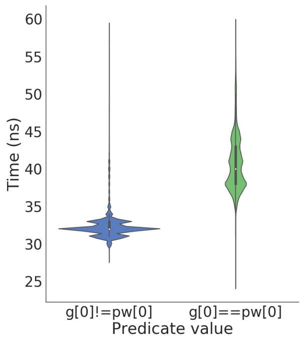
```
public bool check(String guess) {
   bool flag=true, fakeFlag=true;
   for(int i=0; i<guess.len; i++) {
      if (guess[i] != password[i])
         flag = false;
      else
         fakeFlag = false;
   }
   return flag;
}
```

Fig. 1: A naive password-checking method and a "fixed" one.

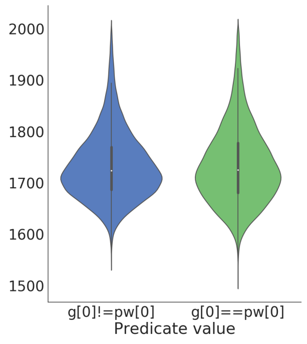
In this paper we present a new type of side-channel vulnerability that is due to optimizations introduced by Java's just-in-time (JIT) compilation mechanism. JIT compilation is present in all modern Java Virtual Machine (JVM) implementations, and is crucial to the performance of Java programs. We show that, if an attacker is able to repeatedly execute a method on a JVM, she can use this capability to trigger JIT optimizations that reduce the execution time of certain execution paths in the method. This enables the attacker to infer predicates about secret inputs passed to the same method. In particular, the attacker can infer values of predicates that correspond to branch conditions in the method body.

Consider the naive password-checking algorithm shown in Figure 1 (left). The guess and a secret password are compared character by character. As soon as there is a mismatch, the algorithm returns false. This early return results in a timing side channel that enables an observer to correlate the execution time of the method with the number of characters matched.

A security-conscious developer might decide that, since the method handles sensitive data, it is worth sacrificing the early return in exchange for a more secure function. They might propose a method like the one shown in Figure 1 (right). In this new version of check, an equal amount of work is performed regardless of the length of the matching prefix.



(a) With JIT enabled



(b) With JIT disabled

Fig. 2: Execution time of the “fixed” check method.

The side-channel vulnerability appears to have been fixed in the new version of the code. However, the source code written by the developer is not the only factor impacting the execution time of program paths. The runtime environment itself can introduce timing side channels into deceptively secure-looking code fragments when it attempts to optimize paths that it deems “hot”. For example, the JVM tracks how often each branch of a conditional branch instruction is taken, and uses this information when JIT-compiling a method to generate native code favoring the more frequent branch. If an attacker is guessing potential passwords randomly, the probability of missing is much higher than that of matching something. As a result, the *then* branch heats up, and JIT introduces a timing side channel into the supposedly “fixed” version. Figure 2a depicts the clear separability of the method’s execution time distributions when the first character misses (optimized branch) and when it correctly guesses (non-optimized branch) the first character of the password. Figure 2b shows how the side channel disappears when JIT is disabled.

The art of inducing JIT-based timing side channels is not easy to master due to the complex and subtle interactions between different kinds of JIT optimizations, the presence of noise in execution timing, and the imperfect control that an attacker may have over the state of the JVM. In this paper we present a systematic approach for inducing JIT-based side channels that leak the value of a predicate on secret inputs.

Given a method m and a predicate ϕ in the body of m, our approach consists of four steps:

1) **Profiling:** We repeatedly run m on an input value that causes the ϕ branch to be taken, and on one that causes the $\neg\phi$ branch to be taken. This enables us to measure the effect of JIT optimizations on execution time.
2) **Priming:** We run m repeatedly with the same input value to induce JIT optimizations.
3) **Timing:** We time the execution of m for a particular input value.
4) **Inference:** Based on the timing profile learnt during the profiling phase and the execution time value that we observed during the timing phase, we infer the value of the predicate ϕ for the secret input.

We demonstrate that this approach can be used to construct two kinds of JIT-based side-channel attacks. In one class of attacks that we call *volatile-secret* attacks, we assume that we are able to repeatedly trigger execution of m on an input of our choice, but we can trigger the execution of m on the secret input only once. In another class of attacks that we call *persistent-secret* attacks, we assume that we are able to repeatedly trigger execution of m on the secret, and that we can trigger its execution at least once on an input of our choice.

We induce the JIT attacks outlined above by leveraging various JIT optimizations. We identify several *vulnerability templates*, each one based on exploitable JIT optimizations. These templates help us identify branch instructions in the body of m that are amenable to JIT-based attacks, and provide a recipe to find the right priming parameters for each case.

We implement the approach discussed above and apply it to widely used classes from the Java standard library. We use symbolic execution to generate suitable priming values for our experiments. Since the timing distribution of different execution paths can overlap, we may not always reach full certainty about the value of the predicate, even if we induce a strong side channel. We use the conditional entropy between the timing information and value of the predicate to quantify how much information is leaked about the predicate. Our experiments demonstrate that JIT attacks are feasible for many library functions from said classes, and show how a malicious user can learn values of predicates about secret inputs using JIT-based attacks.

Our contributions in this paper are:

- Definition and demonstration of a new class of timing side-channels based on JIT optimizations.
- Two attack models for learning predicates about secret inputs based on JIT-borne side channels.
- Five vulnerability templates to identify code fragments amenable to JIT attacks and to guide the process of inducing them.
- Techniques to induce JIT-based side channels and infer predicate values based on them.
- Experimental evaluation of JIT-based side channels on widely used methods in Java standard library.

The rest of the paper is organized as follows: In Section II we review Just-In-Time compilation for Java. In Section III we present our technique for inducing JIT-based side channels. In Section IV we describe our implementation and experimental setup. In Section V we discuss the results of our experimental evaluation. In Section VI we discuss related work. In Section VII we present our conclusions and ideas for future work.

II. Just-in-time compilation of Java bytecode

In this section we review the essential characteristics of the Java Virtual Machine and its Just-In-Time compilation mechanism. We describe the main JIT optimizations leveraged in our work, and provide references to further technical information.

A. Java and the HotSpot Java Virtual Machine

The Java platform includes the Java Language Specification and the Java Virtual Machine Specification [8]. The official reference implementation of the JVM Specification is the HotSpot virtual machine [9] that we use in this work. HotSpot was started by Sun [10] and is now maintained by Oracle. Since 2006, its codebase is open source through the OpenJDK project [11]. There are only a few subtle differences between the Oracle and OpenJDK development kits, and no significant differences between Oracle HotSpot and OpenJDK HotSpot.

The javac tool compiles Java source code to Java bytecode, which is then executed by the Java virtual machine. Executing bytecode requires translating it to native machine code for the platform at hand (e.g., Intel x86). The easiest way to achieve this is *interpretation*, i.e., translating bytecode instructions to native instructions as they are encountered, which is simple but results in slow performance. When a method is costly and executed often, it may make economic sense for the virtual machine to take a moment and permanently compile it into optimized, reusable machine code that will run faster.

B. Just-In-Time (JIT) compilation

Based on the general observation that most of the execution time is typically spent executing a small fraction of the code, the HotSpot JVM uses runtime profiling to detect "hot spots" that are worth feeding into an optimizing compiler. In fact, modern versions of the JVM attempt to dynamically adjust the optimization level (and thus the compilation overhead) of each method in order to maximize the return on investment.

The *client-mode JIT compiler* (C1) is a fast bytecode-to-native compiler that only performs a small set of lightweight optimizations. It thus minimizes compilation overhead at the expense of runtime efficiency. It was originally designed for the "client" flavor of the JVM, which favors fast launch times.

The *server-mode JIT compiler* (C2) is a slow, but highly optimizing bytecode-to-native compiler that performs a wide spectrum of costly optimizations. Originally designed for the "server" JVM [12], it generates the fastest native code at the expense of higher compilation time and memory overheads.

Starting with Java 7, the JVM supports *tiered compilation* mode, which combines the best of both modes. In the server mode, the VM used the bytecode interpreter to collect profiling information about each method. In tiered mode, the VM uses the C1 compiler to generate compiled versions of methods that collect profiling information about themselves. Based on that information, it may decide to recompile a method with the C2 compiler. C1-compiled code is much slower than C2-compiled code, but substantially faster than interpreted code; thus, the tiered VM runs the program faster during the profiling phase. The tiered scheme offers quick startup times like client mode, and can also achieve better peak performance than server-only mode because the faster profiling phase allows a longer period of profiling, which may result in better optimization [13]. In tiered mode, the C1 and C2 compilers are used as the basis of a scheme that includes five tiers (levels of compilation) ranging from purely interpreted (L0) to fully optimized (L4).

C. Main JIT compilation techniques exploited in this work

The JIT compilation scheme includes many techniques and optimizations. In this section we briefly describe those that constitute the basis for the vulnerability templates that we will present in Section III.

1) Method compilation: The HotSpot compilation policy relies on runtime profiling metrics and makes runtime decisions. One of the main factors that affect when and how methods are compiled are the method invocation counters that track how often each method is invoked. When the appropriate thresholds are reached [14], the method may be scheduled for compilation, or for recompilation at a higher tier. Another factor that can promote the [re]compilation of a method are back-edge counters that track how often backward jumps (typically associated with loops) are taken.

2) Branch prediction: JIT branch prediction uses counters to track how often each branch of a conditional branch instruction is taken. When a method is compiled, this information is exploited to generate native code where the most frequently taken branch appears first, thus avoiding a jump instruction. The savings are amplified in the case of loops. Although this optimization is independent of CPU-level branch prediction, it can achieve positive synergy with it.

3) Optimistic compilation: When a method is C2-compiled, if the counters show a heavy enough bias toward one side of a conditional, the other branch is not compiled at all—its code is simply removed. The resulting optimized code assumes that the rarely-taken branch will never be taken, and the missing code is replaced by a trap that is triggered if the rare branch is taken. This is known as an *uncommon trap*. If and when the rare branch is taken, the uncommon trap handler must *de-optimize* the method and replace the optimistically compiled version with a more conservative, slower version.

4) Method inlining: If a method is deemed small enough, it may be inlined into its callers, thus avoiding the overhead of a method call. This deceptively simple-looking optimization is in fact one of the most complex ones in the scheme, as it interacts with others in nontrivial ways. For instance, when m calls m', inlining m' into m can impact ulterior optimizations of m, and the same effect may cascade to deeper levels. While none of our exploits is based solely on inlining, we do use this optimization in combination with other ones (see Section V).

5) Other optimizations: HotSpot JIT compilation features many other optimizations, e.g., loop unrolling, escape analysis, dead code elimination, etc. Some are essentially akin to those present in modern static optimizing compilers, while many others are truly adaptive in nature and can only be performed in a context where they may be de-optimized as needed. For further details we refer the reader to the documentation [9].

III. Attack models and templates

In this section we present our techniques for inducing JIT-based side channels. We introduce the *volatile-secret* and the *persistent-secret* attack models. Each model comprises a set of assumptions that stipulate what actions can be triggered by the attacker, which input arguments are controllable, and what can be timed.

Given an attack model, JIT-based vulnerabilities could be manually crafted in countless ways by exploiting peculiarities of the code under test and complex combinations of multiple optimization types. To systematize the process of inducing vulnerabilities and render it more amenable to automation we present several *vulnerability templates*. The templates capture the necessary code patterns for each kind of vulnerability, and provide a recipe for inducing it.

A. Branch instructions and associated predicates

Let m be a Java method that contains at least one conditional branch instruction. In the simplest terms, selecting a branch instruction for dynamic side-channel analysis amounts to selecting an if statement. In practice, many Java constructs give rise to branching behavior: conditional statements like if and switch, expressions like `c?x:y`, the guard conditions of while and for loops, etc. When translated to bytecode, such constructs give rise to conditional branch instructions.

A conditional branch instruction has an associated *predicate* ϕ, which is the branch condition expressed in terms of the input. The path of execution up to the branch instruction can be characterized with a *path constraint* π. Thus, an input to method m satisfies $\pi \wedge \phi$ if and only if it reaches the conditional branch instruction and takes the *then* branch, and it satisfies $\pi \wedge \neg\phi$ if and only if it reaches the instruction and takes the *else* branch. In contexts where no ambiguity arises, we refer to the conditional branch instruction by its associated predicate. We address the problem of introducing JIT-based side-channel vulnerabilities that enable an attacker to learn the value of the predicate ϕ on an input to method m.

B. The volatile-secret attack model (VSAM)

The first model we introduce is the *volatile-secret attack model* (VSAM). In this model, we assume that we can repeatedly trigger m on an input value of our choice and time a single subsequent call to m on a secret value s. Our goal is to infer whether s does or does not satisfy ϕ. To improve our chances of success, we leverage our ability to repeatedly trigger m on a value of our choice to *prime* the JVM. By *priming* we mean the action of repeatedly executing m with inputs that exercise certain paths with respect to ϕ, in an attempt to "heat up" the JVM. The goal of our priming under VSAM is to encourage the JVM into a state where the execution time of the call to m on s is correlated with the value of the predicate ϕ on s. This is done by choosing a priming value to induce heavier optimization on one branch.

Inferring the value of ϕ on s from the execution time of the call to m requires developing a statistical profile of the execution times of m for both values of ϕ after priming with a chosen priming value. This in done in our *profiling phase*. Developing a statistical profile also benefits us twofold. First, time measurements are affected by nondeterminism from various sources, from the inevitable system noise to minor variations in runtime decisions made by the JIT compiler as to which optimizations to apply and in what order. The statistical nature of the profile allows us to effectively handle such noise. Secondly, the assumption that the attacker has complete control over the JVM is often unrealistic. When we build a statistical profile, we can simulate an environment where some proportion of triggers of m is outside the control of the attacker. By *priming with an α distribution* (with respect to ϕ) we mean priming m with inputs that will exercise one branch (ϕ) with probability α, and the other branch ($\neg\phi$) with probability $1-\alpha$. When we build a statistical profile, we will prime with $\alpha < 1$ to simulate an environment where m is occasionally triggered on some input exercising the opposite path to the one we have chosen to heat up.

Using this profile, we are able to make an inference about the value of ϕ on s from the execution time of the call to m.

The pseudocode in Algorithm 2 outlines the above process. Here, the two priming input values $p_\phi, p_{\neg\phi}$ are chosen such that (a) both reach the conditional branching instruction of interest and (b) p_ϕ takes the ϕ branch whereas $p_{\neg\phi}$ takes the $\neg\phi$ branch. The test values t_ϕ and $t_{\neg\phi}$ are chosen from the set of possible secret values taking the ϕ branch and the $\neg\phi$ branch respectively to generate representative timing information for a secret value taking the corresponding branch. The priming amount n is the total number of calls to the method m in the PrimeAndTime subalgorithm (see Algorithm 1) and the profiling amount N is the number of times the priming and then timing subroutine is repeated during profiling in order to generate a statistical profile robust to noise.

Choice of priming input: The choice of input used for priming can greatly impact the success of correctly inferring ϕ on s. In particular, choosing $p_\phi, p_{\neg\phi}, t_\phi$, and $t_{\neg\phi}$ to satisfy the same path constraint π leading to ϕ as s improves the chances that our statistical profile accurately models the execution time information we might receive from a call to m on s. We describe in Section IV-B how we can choose values for $p_\phi, p_{\neg\phi}, t_\phi$ and $t_{\neg\phi}$ intelligently if we assume we know the path constraint π of s. Likewise, knowing certain properties of the execution of m on s after ϕ has been passed and choosing t_ϕ and $t_{\neg\phi}$ to match those when possible also improves our probability of determining the value of ϕ on s. Though both of these seem to be strong assumptions, in our experimental section we argue that they can often be satisfied in practice.

C. Vulnerability Templates for VSAM

We provide a series of vulnerability templates for the purposes of easy identification of code susceptible to a JIT-based side channel and for systematic understanding of the parameters necessary for the side channel to be exploitable under VSAM.

Each vulnerability template has

- a particular kind of optimization that it exploits (e.g., optimistic method compilation)
- a code pattern, e.g., that a method m' must be called somewhere in one branch of ϕ and may not be called in the other branch

- a recipe that guides the search of suitable parameters for priming: what distribution to use, how to determine the number of iterations, etc.

*1) Branch prediction (*TBRAN*):* This template exploits the JIT branch prediction mechanism, which generates slightly more efficient native code by placing the branch that is taken more often first, thus reducing the number of jump instructions (see II-C2). This template can be applied to any conditional statement, but the imbalance that it introduces is small, so that it may only be observable in some kinds of methods. This template works best in situations where the conditional is enclosed in a loop (which amplifies the small difference), or in small methods, where the small difference achieved is significant w.r.t. the cost of the rest of the method. The amount of priming must be sufficiently high that JIT deems generating the more efficient native code a worthwhile effort. Additionally we must control a high enough fraction of the calls to m to ensure that the ratio of choices made at ϕ is biased in favor of our priming input.

*2) Optimistic method compilation (*TOPTI*):* This template exploits the optimistic compilation mechanism (see II-C3), a much more aggressive flavor of branch prediction which does not even generate the code for the uncommon branch. More precisely, it exploits the fact that when the uncommon branch *is* taken, the method is de-optimized. Our priming needs to ensure that (a) the method is called enough times to be compiled at the C2 level, and that (b) by the time that happens, the conditional of interest has been taken in one direction the vast majority of the times it has been executed. This template is applicable to any branch conditional where the choice on ϕ means that some instructions are never executed.

*3) Method compilation (*TMETH*):* This template exploits the difference in speed of execution between interpreted and compiled (or between C1-compiled and C2-compiled) code. It requires that the selected conditional in method m performs a call to some other method m' in one of its branches, but not in the other one. Priming must ensure that the branch that calls m' is executed a sufficiently high number of times, so that m' is compiled to a faster version. The speeding up of m' thus causes an observable imbalance in the timing of m.

D. The persistent-secret attack model (PSAM)

The second model we introduce is the *persistent-secret attack model*. In this model, we assume that we can repeatedly trigger m on a secret value s and time a single subsequent call to m on a value t of our choice. Our goal is again to infer whether s does or does not satisfy ϕ. As in VSAM, we prime the JVM to increase the probability that we can correctly infer the value of ϕ on s. In PSAM the priming input is the secret value s. We choose a priming amount n and a test value t such that the execution time of m on t after priming with n triggers to $m(s)$ will reveal information about the value of ϕ on s. For this to be the case, the optimizations induced by priming with a value taking the ϕ branch must be different from those induced when priming with a value from the $\neg\phi$ branch. Additionally, these differences must result in observably different execution times of the $m(t)$ call between the two kinds of priming values.

We develop a statisical profile to reliably perform inference when presented with the execution time of the method m on t after priming with the unknown value s. We choose two priming inputs p_ϕ and $p_{\neg\phi}$ executing the ϕ and $\neg\phi$ branches respectively and generate a statistical profile of the execution time of m on t for both priming inputs. Once again, we introduce the ratio α to simulate an environment where some triggers to m are outside of the attacker's control. The profiling and subsequent inference specific to PSAM is given in Algorithm 3. The strong assumption we make is that the timing profile generated using p_ϕ and $p_{\neg\phi}$ is representative of what we can expect to see when priming using s. As in VSAM, this assumption becomes more believable if we assume that the path constraint π of p_ϕ and $p_{\neg\phi}$ matches that of s.

```
input : $n$ (priming amount), $\alpha$ (priming ratio),
        $p_{\text{more}}$, $p_{\text{less}}$ (priming inputs), $t$ (test input)
output: timing in nanoseconds of the last call to $m$
PrimeAndTime($n$, $\alpha$, $p_{\text{more}}$, $p_{\text{less}}$, $t$)
numItersBothSides ← $2(n - n \cdot \alpha)$;
numItersRemaining ← ($n$ − numItersBothSides);
for $i \leftarrow 1$ to numItersBothSides do
    if $i$ is odd then
        trigger $m(p_{\text{more}})$;
    else
        trigger $m(p_{\text{less}})$;
    end
end
for $i \leftarrow 1$ to numItersRemaining do
    trigger $m(p_{\text{more}})$;
end
timingOfTestInput ← time $m(t)$;
return timingOfTestInput;
```

Algorithm 1: PrimeAndTime pseudocode

```
input   : $N$ (profiling amount), $n$ (priming amount), $\alpha$ (ratio),
          $p_\phi$, $p_{\neg\phi}$ (priming inputs), $t_\phi$, $t_{\neg\phi}$ (dual test inputs)
unknown: $s$ (secret)
$v_\phi$, $v_{\neg\phi}$ ← two empty vectors to store timing profiles;
for $i \leftarrow 1$ to $N$ do
    $v_\phi$.append( PrimeAndTime($n$, $\alpha$, $p_\phi$, $p_{\neg\phi}$, $t_\phi$) );
end
for $i \leftarrow 1$ to $N$ do
    $v_{\neg\phi}$.append( PrimeAndTime($n$, $\alpha$, $p_\phi$, $p_{\neg\phi}$, $t_{\neg\phi}$) );
end
timingOfSecretInput ← PrimeAndTime($n$, 1.0, $p_\phi$, null, $s$);
leakageEstimation ← InferPredicate($v_\phi$, $v_{\neg\phi}$, timingOfSecretInput);
```

Algorithm 2: VSAM attack pseudocode

E. Vulnerability Templates for PSAM

All vulnerability templates that apply to VSAM are applicable to PSAM as well. Additionally:

*1) Self-compilation via back-edge counters (*TSELF*):* This template exploits method compilation due to back-edge counters rather than method invocation counters (see II-C1). It does not require m to call another method m'. Instead, m itself is (or is not) compiled (or is compiled to a different level of optimization) depending on whether the back-edge

```
input    : $N$ (profiling amount), $n$ (priming amount), $\alpha$ (ratio),
           $p_\phi$, $p_{\neg\phi}$ (profiling priming inputs), $t$ (single test input)
unknown: $s$ (secret)
$v_\phi$, $v_{\neg\phi}$ ← two empty vectors to store timing profiles;
for $i \leftarrow 1$ to $N$ do
    $v_\phi$.append( PrimeAndTime($n$, $\alpha$, $p_\phi$, $p_{\neg\phi}$, $t$) );
end
for $i \leftarrow 1$ to $N$ do
    $v_{\neg\phi}$.append( PrimeAndTime($n$, $\alpha$, $p_{\neg\phi}$, $p_\phi$, $t$) );
end
timingOfSecretPriming ← PrimeAndTime($n$, 1.0, $s$, null, $t$);
leakageEstimation ← InferPredicate($v_\phi$, $v_{\neg\phi}$, timingOfSecretPriming);
```

Algorithm 3: PSAM attack pseudocode

counters are sufficiently high. This method is exploitable when ϕ impacts the number of back edges (jumps to previous code) traversed. This is commonly due to ϕ impacting the number of iterations of a loop. The priming amount must be chosen to induce the difference between the optimization level of m according to the two priming scenarios. The ideal test value t for this vulnerability template is one for which the method m is expensive – making the difference in execution time between its differently compiled versions more apparent.

*2) Method compilation with timing of the inner method (*TMETH-TI*):* This template is an extension of the TMETH template of VSAM. This template applies to any branch instruction where a method m' is called more frequently in one branch of ϕ than the other. The case where m' is never called in one branch is an extreme case of this scenario. The priming amount is chosen so that the level of compilation of m' is different across the two priming scenarios. The ideal test value for this case is one in which calls to m' are expensive. Under an additional assumption that the attacker can call and directly time m', she can obtain the execution time of m' without interference from the rest of method m. This enables more reliable inference.

IV. Implementation and Experimental Setup

In this section we describe our experimental subjects, setup, and decisions made with respect to experimental evaluation.

A. Source of experimental subjects

We evaluated our technique on the **java.math.BigInteger**, **java.lang.Math** and **java.lang.String** classes from the Java standard library (JDK 8, rev. b132) [15]. We removed methods with no conditional branches, native methods which are not written in Java, duplicates except for minor type differences (e.g., float vs. double), and some which, though not literal duplicates, had essentially isomorphic control-flow structures. For the remaining methods we applied our technique and chose branch instructions that satisfied the most relevant templates. In the following sections we present a selection of our results featuring the cases (both successful and unsuccessful) that we found most interesting and relevant.

B. Using symbolic execution to find priming inputs

Let m be the method under analysis. Suppose a conditional branch instruction within m with predicate ϕ is selected per

TABLE I: Priming distributions used in our experiments

	VSAM α-ratio	PSAM α-ratio
TOPTI	0.998	0.998
TMETH	0.950	0.950
TBRAN	0.900	0.950
TSELF	n/a	0.950

one of the templates. To evaluate our approach we need to generate suitable priming input values for m that reach ϕ.

In particular, as explained in Section III, we want to generate a pair of priming input values for m that share a common execution path prefix until they reach ϕ. More precisely, we want to find a path constraint π that leads to ϕ, and for which there exists a pair of input values $\langle p_\phi, p_{\neg\phi}\rangle$, both satisfying π, such that p_ϕ satisfies $\pi \wedge \phi$, and $p_{\neg\phi}$ satisfies $\pi \wedge \neg\phi$.

Finding these manually can be a tedious and error-prone process. It is desirable to automate priming input generation, which significantly speeds up the task of exploring multiple methods and multiple predicates within them to find instances of vulnerability templates. We use the Symbolic PathFinder (SPF) [16] tool to symbolically execute the Java method under analysis. The tool runs m on symbolic inputs that represent multiple possible concrete inputs. Variables are represented as constraints which are solved using the Z3 constraint solver [17]. During SPF's exploration of the symbolic execution tree, we listen to the events as they occur and look for a path that simultaneously satisfies the aforementioned conditions. Note that this is not just a matter of reaching ϕ: backtracking may be needed even after reaching ϕ, for instance, if a candidate π is found for which one branch is satisfiable but the other one is not. If a suitable path is found, we immediately end the symbolic exploration and save the path constraint π, its two extensions $\pi \wedge \phi$ and $\pi \wedge \neg\phi$, and a pair of witnesses $\langle p_\phi, p_{\neg\phi}\rangle$ that can be used as input values for priming.

C. Computing information leakage via conditional entropy

For each experimental subject we ran 1000 iterations and, on each iteration, we primed the system as described in each of the next subsections and timed the subsequent call to the method under test. From this data we computed the conditional entropy between the value of the predicate and the observed timing distribution. This tells us how many bits of information about the value of $\phi(s)$ we can expect to be leaked from a single time measurement. Since the value of ϕ encodes one bit of information, a conditional entropy value of 0.0 means no leakage, whereas 1.0 means that we can determine the value of $\phi(s)$ from one timing observation with complete certainty.

D. Using priming distributions to simulate noisy triggering

As discussed in Section III-B, the α ratio is an experimental parameter to account for the fact that, in a realistic scenario, we may not have exclusive control over the state of the JVM. Table I shows the distributions that we associated with each template under each model.

E. VSAM experiments

For VSAM cases, we used SPF to generate two witnesses for priming. We then generated two sets of possible secret inputs satisfying the same path constraint π as the witnesses. The secret inputs in the first set execute the $\pi \wedge \phi$ path, while those in the second set take the $\pi \wedge \neg\phi$ path. We primed the JVM with the SPF-provided witnesses using the priming ratio α indicated by the template.

For TMETH and TOPTI cases we determined the number of priming iterations as follows: Starting with an initial guess, use the JITWatch tool [18] to determine whether the optimization has occurred. If not, increase the number of iterations until it does. For TBRAN cases we tried priming {1000, 10000, 50000, 100000} times and kept the value that maximizes leakage.

For evaluation, we repeated the following 1000 times. We primed the system as described above, then timed a call to the method on a randomly chosen secret value executing the ϕ branch. Then we performed another 1000 iterations of the experiment, now timing a call to the method on a randomly chosen secret value executing the $\neg\phi$ branch. From this data we computed the leakage as explained in IV-C.

In addition to the aforementioned, we also re-executed all experiments (and recomputed the leakage each time) for the following three priming scenarios:

1) Reversed priming: We re-ran all experiments with a ratio $\overline{\alpha} = (1 - \alpha)$ instead of α. In other words, if the ϕ branch was the one more heavily primed in the original experiment, the $\neg\phi$ branch now is, and vice versa. This evaluates whether priming more strongly in favor of that particular branch is critical to the success of the technique for that experimental subject, or whether either of the two branches would suffice.

2) Even priming: We re-ran all experiments with a fixed ratio $\alpha = 0.5$, i.e., the same amount of priming on both branches. This evaluates the importance of the imbalanced priming ratio in introducing side channel, as opposed to the more general, overall heating up of the whole method.

3) No JIT: We re-ran all experiments with JIT disabled. This evaluates the existence of a static (traditional, source-code level) side-channel vulnerability, which our use of JIT could augment or mitigate. We still used a fixed, very small amount of even priming (50 calls on each branch) to avoid artificial noise from initial class/method loading delays.

F. PSAM experiments

For cases evaluated under PSAM, we again used symbolic execution to generate two witnesses for priming. We made the assumption that the results of priming with a given witness are representative of the results of priming with any value from the same equivalence class as that witness, that is, any value that satisfies the same path constraint as the witness and executes the same branch of ϕ.

For each case, we determined the number of priming iterations in the same way as for VSAM (see IV-E), starting with an initial guess and using JITWatch to guide the search.

For evaluation, we repeated the following experiment. We primed the system (with priming parameters obtained as described above) in favor of the witness for the ϕ branch, and then timed a subsequent call on a chosen test input t. We manually chose t such that the difference between the optimization levels after the two types of priming would be observable. We experimented again by priming the system with the same parameters as before, but in favor of a witness for the $\neg\phi$ branch, and then timed a subsequent call on the same test input t. Each experiment was repeated 1000 times. From this data we computed the leakage as explained in IV-C.

G. 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.

V. EXPERIMENTAL RESULTS

Table II and Table III summarize our results for VSAM and PSAM, respectively. For each set of experiments we report the method name, location of the selected branch instruction in the class source code [15] (rev. b132), the template that was used, other templates (if any) that also arose accidentally, and the priming parameters used. For VSAM and PSAM we report the amount of information leaked about the predicate value. For VSAM we also report the information leakage of three additional experimental sets (reversed priming ratio $\overline{\alpha}$, even priming, and JIT disabled) as described in Section IV-E.

A. Inferring the value of a predicate

Inferring the value of a single predicate about a secret may seem less informative than extracting the whole secret itself. Nevertheless, examples of the predicates that we inferred in our experiments show how they pose real security threats. For example, in BigInteger.min, Math.max, and Math.min, we learn which of two inputs is larger. When one of these inputs is secret and one is controllable, this enables a binary search attack to find the value of the secret! Similarly, the predicate in the String.equals, String.compareTo, and String.startsWith cases denotes whether the first characters of two different strings match. If similar code is used in a login module, an attacker could exploit the side channels that we induce to determine the first character of a secret password. The side channel could then be iterated to continue inferring the password character by character.

B. Assumptions on path constraints

We evaluate the VSAM cases with secret test values that share the same path constraint as the priming values. In general, the assumption that the secret will share the same path constraint as the priming value is a strong one. However in our experiments, this assumption often amounted to ensuring that the input satisfies certain sanity checks. For example, that the input is not null; that it is not an extreme value such as NaN, positive or negative infinity; or that the length of two strings being compared is equal. Satisfying such conditions was commonly the only requirement for the inputs to have the same

TABLE II: Experimental results for VSAM

Method name	Branch instruction	Template (applied, arisen)	Priming amount	Priming ratio (α)	Leakage under α	Leakage under $\overline{\alpha}$	Leakage 0.5 | 0.5	Leakage w/o JIT
java.math.BigInteger								
min	line 3477	TOPTI	100,000	0.998	**1.00**	**1.00**	0.02	0.06
valueOf	line 1085	TBRAN	10,000	0.900	**0.52**	**0.16**	0.10	0.03
shiftLeft	line 2908	TMETH	10,000	0.950	**0.99**	**0.95**	0.75	0.79
java.lang.Math								
max	line 1316	TBRAN	10,000	0.900	**0.28**	**0.25**	0.04	0.03
ulp	line 1443	TMETH	50,000	0.950	**0.05**	**0.05**	0.02	0.25
nextAfter	line 1926	TOPTI	100,000	0.998	**1.00**	**0.89**	0.02	0.03
min	line 1350	TOPTI	100,000	0.998	**0.03**	**0.01**	0.02	0.03
java.lang.String								
equals	line 976	TOPTI, TBRAN	100,000	0.998	**0.44**	**0.04**	0.12	0.04
compareTo	line 1151	TOPTI	100,000	0.998	**0.99**	**0.03**	0.02	0.20
startsWith	line 1400	TBRAN	1,000	0.900	**0.46**	**0.16**	0.21	0.25

TABLE III: Experimental results for PSAM

Method name	Branch instruction	Template (applied, arisen)	Priming amount	Priming ratio (α)	Leakage under α
java.math.BigInteger					
mod	line 2402	TMETH	10,000	0.950	**0.33**
mod	line 2402	TMETH-TI	10,000	0.950	**1.00**
and	line 3054	TSELF	500	0.950	**1.00**
java.lang.Math					
scalb	line 2287	TSELF	5,000	0.950	**0.58**
java.lang.String					
trim	line 2857	TSELF	5,000	0.950	**1.00**
replace	line 2060	TSELF, TBRAN	2,000	0.950	**0.92**
replace	line 2060	TBRAN	2,000	0.950	**0.66**
Constructor	line 250	TMETH-TI	500	0.950	**0.08**

path constraint. This observation also makes more reasonable the PSAM assumption that the values we choose for priming during the profiling phase have the same path constraint as the secret priming value. Even in the few cases where the path constraint was more nuanced, such as String.replace, it was still very reasonable (the character to be replaced must not be the same as the one it will be replaced by).

*C. Optimistic compilation (*TOPTI*)*

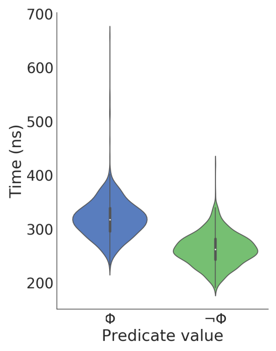
When optimistic compilation could be induced, the uncommon trap execution time was always at least two orders of magnitude higher (e.g., see Fig. 3b), resulting in very reliable learning of $\phi(s)$. Our high-leakage results for BigInteger.min, Math.nextAfter, and String.compareTo were obtained in this way. For the other two cases, Math.min and String.equals, our priming did not succeed in inducing optimistic compilation. In Math.min, this was due to inlining: Math.min is so small that it immediately gets inlined into its caller (i.e., into our experiment driver). Optimistic compilation could still be induced into the inlined copy of Math.min, but would not be exploitable in other inlined copies. For String.equals, we could not induce optimistic compilation due to a combination of two facts: (i) optimistic compilation requires an extremely lopsided history at the time of C2-compilation, and (ii) String.equals is triggered too frequently by other parts of our experiment driver. Hence, this template is suitable for contexts where the attacker has nearly-exclusive control over the triggering of m. In constrast, the String.compareTo method, which has an almost identical structure to String.equals with respect to the selected predicate and its branches, was much more amenable to an optimistic compilation exploit due to its less frequent usage elsewhere.

Despite our inability to induce an optimistic compilation of String.equals, we still achieved very sizeable leakage in this method thanks to branch prediction, which does not require a history as strongly lopsided as optimistic compilation.

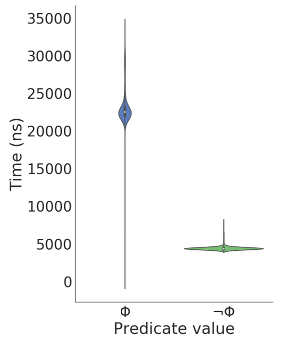
Note that when an attacker succeeds in inducing optimistic compilation, the first call to m that takes the uncommon branch will trigger the uncommon trap, and de-optimization will only take place once. This means that the attacker must trigger and time m on the secret input before some other user triggers and thus "spoils" the uncommon trap through their own calls.

*D. Method Compilation (*TMETH*)*

Our high-leakage result for BigInteger.shiftLeft exemplifies the potential of TMETH under VSAM. In shiftLeft, each of the two branches of ϕ calls a different method. With JIT disabled, the execution time *does* leak information about which



(a) String.equals



(b) String.compareTo

Fig. 3: Execution time distributions after priming for the methods (a) String.equals and (b) String.compareTo.

branch was taken. This is not surprising, as it is expectable that the unoptimized versions of two different methods would be distinguishable. What we wish to emphasize is that any of the two callee methods can be made observably faster than the other through the appropriate priming. Moreover, both of these priming versions result in stronger side channels than those that occur with JIT disabled or with an even priming distribution. This demonstrates how strongly the execution time of a path can vary depending on how aggressively the methods called along that path are optimized.

In Math.ulp, we observed a scenario in which a method was called in one branch of ϕ but not the other, motivating us to apply the TMETH template. We thought that by compiling that method, we might significantly reduce the execution time of its branch. This was not the case. The method that we aimed at (and succeeded at) compiling was an extremely inexpensive, constant-time method. Thus the timing of the branch containing it did not change significantly with compilation, and its cost did not fall below that of the other branch. The degree to which an application of TMETH can impact the execution time is bounded by the degree to which compilation can speed up the method called in the heated-up branch.

The difference between the two results that we report for BigInteger.mod is due to the improved reliability of timing data when the method whose execution we time is not m itself, but rather its callee method m' whose compilation we aimed to induce (TMETH-TI). This requires stronger assumptions (the callee method m' must be triggerable and timeable by the attacker), but it can substantially increase the leakage.

However, not even obtaining such isolated timing information can enable a side channel when the computation done by m' is trivial. This is the case for the String constructor we analyzed, which builds a string based on a sequence of Unicode code points. Though we succesfully found priming values inducing different levels of optimization in its callee method m', hardly any leakage resulted. This is due to m' performing extremely efficient constant time compuation, making the difference in efficiency between its compiled and un-compiled states indiscernible.

E. Branch Prediction (TBRAN)

Branch prediction introduces considerably smaller timing differences than other templates (e.g., see Fig. 3a). Nevertheless, it can still sometimes be exploited to great effect. BigInteger.valueOf, String.startsWith, and Math.max are examples of methods that are small enough that the effect of branch prediction is observable over the computational noise of the method. Whether or not the branch condition is looped over can also impact the observability of the side channel. In PSAM, where we can choose the test value, such a looping construct may enable the choice of a test value for which the effects of branch prediction are multiplied, i.e., the branch prediction is repeatedly correct or incorrect across the iterations of the loop. String.replace (discussed in the next section due to its interaction with the TSELF template) is an example of this scenario.

F. Self Compilation (TSELF)

The TSELF vulnerability template is specific to PSAM. In all cases where we tried to apply TSELF, we were successful in finding priming amounts such that the method was compiled to different levels of optimization across the two priming options. In the BigInteger.and and String.trim cases, it was easy to find a test value that made the method call expensive enough for differing levels of compilation to be observable. This is due to the large number of potential loop iterations within these methods. This was not the case in Math.scalb, where the maximum possible number of loop iterations is four. The strength of a side channel introduced by TSELF is thus linked to how expensive the method is question can be made by suitably chosen test values.

In the case of String.replace, we have an interesting example of the interaction between side channels resulting from self compilation and branch prediction. Once again, we succeeded in inducing differing levels of optimization between the two priming options for the same priming amount. However, while priming with that aim in mind, we also induced a side channel based on branch prediction. The execution time of m on t is thus not only based on the compilation level of m but also on whether the execution path of t is favored or hindered by the branch prediction. Since the priming input from the ϕ branch induced the higher level of compilaton of m, we expected that the timing of the call to m on t would be faster for this priming value. When we choose t to benefit from the branch prediction induced by priming on the ϕ branch, this was the case. This result in shown in the first of our two results on String.replace. However, when we choose a priming value that was hindered by the branch prediction induced through priming on ϕ and favored by that induced when priming on $\neg\phi$, the expected outcome was reversed. The timing of the method call was actually faster under the $\neg\phi$ priming. The unintended branch prediction thus interacted with our intended optimization in a way that inverted our expectations. The results for this experiment are given in the second of our results on String.replace.

VI. Related work

To the best of our knowledge, the idea that JIT could impact and potentially introduce timing channel vulnerabilities was first put forth by Page [19]. Noting that compiled code can differ from source code, he explores the impact of dynamic compilation through a case study on his own Java implementation of a double-and-add-based multiplication program. Because the doubling method is called more frequently than the addition method, it is compiled sooner. If an attacker can obtain a timing profile of the each method called within the multiplication code, they can infer the order of the sequence of doublings and additions performed. Page also proposes some solutions at both the language level and the virtual machine level for removing side channels of this kind.

Our work goes beyond the observation that dynamic compilation *may* introduce side channels by demonstrating how to systematically induce JIT-based runtime-behavior-dependent side channels into the JVM state through controlling the input distribution of a method. We show how to *actively* exploit JIT's focus on optimization to create side channels that enable an attacker to learn predicates about secrets, and experimentally evaluate it on methods from the Java standard library.

In work complementary to ours, Cleemput et al. [20] propose leveraging the statistical profiling information used in dynamic compilation to mitigate timing side-channel vulnerabilities. Starting from a developer-chosen root method, profiling information about the number of back edges taken or method call invocations is collected for each value in a training input set. Based on this process, a set of methods potentially vulnerable to a timing side channel is selected. Control-flow transformations and data-flow transformations are then applied to those methods to reduce their vulnerability to side channels. The control-flow transformations, such as if-conversion, used in their paper would aid in protecting sensitive Java functions from the JIT-based vulnerabilities we introduce. In fact, there is existing work on compiler based strategies for mitigating side-channel vulnerabilities which might be germane to that purpose [21]–[23]. However, none of the solutions that they offer have been integrated into HotSpot, which remains both vulnerable to the kinds of side channels we introduce and the most widely used JVM.

Static Side-Channel Analysis: The problem of statically determining the presence of side channels in software has been widely addressed. Antopoulos et al. [24] and Chen et al. [25] propose techniques to detect imbalanced paths through the control flow graph of a method. More expensive techniques requiring symbolic execution and model counting enable quantifying the amount of information leaked [26] and even synthesizing input so as to maximize the amount of information that can be extracted through the side channel [27], [28]. These approaches rely on a cost model that statically approximates the observable information (such as execution time) along a program path. What our work demonstrates is that such a cost model is insufficient. The execution time of a path depends not only on the instructions along that path but also, and to a great extent, on the state of the JVM. The state of the JVM is in turn influenced by all previous invocations of the code under test. Currently no static approach to side channel detection even attempts to model this complex interaction. In fact, many programs that would be pronounced "safe" by all the static techniques above, including those claiming soundness, would be vulnerable to a JIT-induced side channel captured by one of our templates. Some approaches to side-channel analysis include a dynamic component where runtime information is collected and statistical inference performed [29], [30]. However, none of these approaches consider the space of possible states that the runtime environment might be primed to when collecting this data.

Runtime-based CPU-induced side channels: Branch prediction analysis (BPA) attacks and cache attacks are side-channel attacks which leverage runtime-dependent behavior of CPUs. Cache-based side-channel attacks [3], [31]–[35] have been theorized for years and have increasingly been shown as a powerful technique for recovering sensitive information in practical scenarios. Acıiçmez et al. first demonstrated that the CPU's branch predictor could be leveraged to introduce timing channel vulnerabilities in security-related code [36]–[38]. Since then, the CPU's Branch Prediction Unit has been exploited to introduce various flavors of timing channel vulnerabilities [39]–[41]. While these classes of side-channel attacks focus on runtime behavior due to the state of the processor, we focus on runtime behavior determined by the state of the Java Virtual Machine.

VII. Conclusions and future work

We introduced a new class of runtime-behavior-dependent timing side channels based on execution time imbalances introduced by JIT optimizations. We proposed two attack models and five vulnerability templates, which we evaluated on three well-known Java standard library classes. Our results show that an attacker armed with this knowledge, in a setting satisfying the assumptions of our attack models, can indeed use these techniques to induce side channels that allow her to successfully infer the value of predicates (associated with branch conditions) on secret inputs.

In future work we plan to develop new attack models with fewer assumptions. For instance, we can relax assumptions about the path taken by the secret input to reach the selected branch instruction by expanding our use of symbolic execution to automatically consider *all* such paths up to a certain bound and computing information leakage across them. We also plan to extend our palette of vulnerability templates by considering an even wider set of JIT optimizations and their combinations.

References

[1] J. Friedman, "Tempest: A signal problem," *NSA Cryptologic Spectrum*, vol. 35, p. 76, 1972.

[2] D. Brumley and D. Boneh, "Remote Timing Attacks Are Practical," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251353.1251354

[3] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 605–622.

[4] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 191–205.

[5] N. Lawson. (2009) Timing attack in google keyczar library. https://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/.

[6] (2007) Xbox 360 timing attack. http://beta.ivc.no/wiki/index.php/Xbox_360_Timing_Attack.

[7] (2013) Oauth protocol hmac byte value calculation timing disclosure weakness. https://osvdb.info/OSVDB-97562.

[8] Oracle. The Java Language and the Java Virtual Machine Specifications. https://docs.oracle.com/javase/specs/.

[9] Oracle. The Java HotSpot Virtual Machine at a glance. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html.

[10] JVM Team, "The Java HotSpot Virtual Machine," Technical White Paper, Sun Microsystems, Tech. Rep., 2006.

[11] The HotSpot Group at OpenJDK. http://openjdk.java.net/groups/hotspot/.

[12] M. Paleczny, C. Vick, and C. Click, "The Java HotSpot Server compiler," in *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*. USENIX Association, 2001, pp. 1–1.

[13] Oracle. Java HotSpot virtual machine performance enhancements in Java SE 7. https://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html.

[14] Java HotSpot compilation policy and thresholds. http://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/tip/src/share/vm/runtime/advancedThresholdPolicy.hpp.

[15] OpenJDK: JDK 8 source code (Mercurial repository), tag b132. http://hg.openjdk.java.net/jdk8/jdk8/jdk/.

[16] C. S. Pasareanu, W. Visser, D. H. Bushnell, J. Geldenhuys, P. C. Mehlitz, and N. Rungta, "Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis," *Autom. Softw. Eng.*, vol. 20, no. 3, pp. 391–425, 2013.

[17] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[18] C. Newland. The JITWatch tool. https://github.com/AdoptOpenJDK/jitwatch.

[19] D. Page, "A note on side channels resulting from dynamic compilation," *Cryptology ePrint archive*, 2006.

[20] J. Van Cleemput, B. De Sutter, and K. De Bosschere, "Adaptive compiler strategies for mitigating timing side channel attacks," *IEEE Transactions on Dependable and Secure Computing*, 2017.

[21] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 2009, pp. 45–60.

[22] J. V. Cleemput, B. Coppens, and B. De Sutter, "Compiler mitigations for time attacks on modern x86 processors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 23, 2012.

[23] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity." in *NDSS*, 2015, pp. 8–11.

[24] T. Antopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei, "Decomposition instead of self-composition for k-safety," 2017.

[25] J. Chen, Y. Feng, and I. Dillig, "Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 875–890.

[26] L. Bang, A. Aydin, Q.-S. Phan, C. S. Păsăreanu, and T. Bultan, "String analysis for side channels with segmented oracles," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 193–204.

[27] C. S. Pasareanu, Q.-S. Phan, and P. Malacaria, "Multi-run side-channel analysis using symbolic execution and max-smt," in *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*. IEEE, 2016, pp. 387–400.

[28] Q.-S. Phan, L. Bang, C. S. Pasareanu, P. Malacaria, and T. Bultan, "Synthesis of adaptive side-channel attacks," in *Computer Security Foundations Symposium (CSF), 2017 IEEE 30th*. IEEE, 2017, pp. 328–342.

[29] K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen, "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, 2010, pp. 595–606.

[30] P. Chapman and D. Evans, "Automated black-box detection of side-channel vulnerabilities in web applications," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 263–274.

[31] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," in *European Symposium on Research in Computer Security*. Springer, 1998, pp. 97–110.

[32] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel." *IACR Cryptology ePrint Archive*, vol. 2002, p. 169, 2002.

[33] B. B. Brumley and R. M. Hakala, "Cache-timing template attacks," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2009, pp. 667–684.

[34] C. Percival, "Cache missing for fun and profit," 2005.

[35] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack." in *USENIX Security Symposium*, 2014, pp. 719–732.

[36] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *CryptographersâĂŹ Track at the RSA Conference*. Springer, 2007, pp. 225–242.

[37] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proceedings of the 2nd ACM symposium on Information, computer and communications security*. ACM, 2007, pp. 312–320.

[38] O. Acıiçmez, S. Gueron, and J.-P. Seifert, "New branch prediction vulnerabilities in openssl and necessary software countermeasures," in *IMA International Conference on Cryptography and Coding*. Springer, 2007, pp. 185–203.

[39] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–13.

[40] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside sgx enclaves with branch shadowing," in *26th USENIX Security Symposium, USENIX Security*, 2017, pp. 16–18.

[41] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, D. Ponomarev *et al.*, "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, 2018, pp. 693–707.