JIT Leaks: Inducing Timing Side Channels through Just-In-Time Compilation

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. We show that just-in-time (JIT) compilation, which is crucial to the runtime performance of modern interpreted languages, can introduce timing side channels in cases where the input distribution to the program is non-uniform. Such timing channels can enable an attacker to infer potentially sensitive information about predicates on the program input.**

We define three attack models under which such side channels are harnessable and five vulnerability templates to detect susceptible code fragments and predicates. We also propose profiling algorithms to generate the representative statistical information necessary for the attacker to perform accurate inference.

We systematically evaluate the strength of these JIT-based side channels on the java.lang.String, java.lang.Math, and java.math.BigInteger classes from the Java standard library, and on the JavaScript built-in objects String, Math, and Array. We carry out our evaluation using two widely adopted, open-source, JIT-enhanced runtime engines for the Java and JavaScript languages: the Oracle HotSpot Java Virtual Machine and the Google V8 JavaScript engine, respectively.

Finally, we demonstrate a few examples of JIT-based side channels in the Apache Shiro security framework and the GraphHopper route planning server, and show that they are observable over the public Internet.

I. Introduction

Cyber-attacks that steal 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]. Exploitable timing side channels have been found in Google's Keyczar Library [5], the Xbox 360 [6], implementations of RSA encryption [2], the open authorization protocol OAuth [7], and most modern processors [8], [9]. These vulnerabilities highlight the need for preemptive discovery of side-channel vulnerabilities.

This material is based on research sponsored by NSF under grant CCF-1817242 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.

```
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.

We present a new class of side-channel vulnerabilities due to the optimizations introduced by just-in-time (JIT) compilation. We focus on the HotSpot JVM and Google V8 runtime engines, but any JIT-enhanced runtime is similarly susceptible.

We show that if the input distribution to a program is non-uniform, the JIT compiler state will be primed to favor certain paths, resulting in optimizations that reduce their execution time. This can introduce timing side channels even in programs whose resource consumption has been carefully balanced.

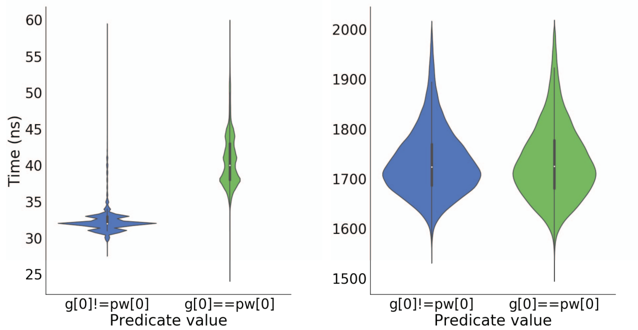
II. An Overview of JIT-Based Side Channels

Consider the naive password-checking method shown in Fig. 1 (left). In this Java method, the password and a guess of matching length are compared character-wise. As soon as there is a mismatch, false is returned. This early return results in a timing side channel that enables an observer to correlate the execution time 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 Fig. 1 (right). In this new version of check, the same amount of work is performed regardless of the length of the matching prefix.

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

DOI 10.1109/SP40000.2020.00007



(a) With JIT enabled (b) With JIT disabled

Fig. 2: Execution time of the "fixed" check method.

	IPM Induced Priming Model	**NPM-LTB** Natural Priming Model Learning Typical Behavior	**NPM-LAB** Natural Priming Model Learning Atypical Behavior
Who primes the runtime?	Attacker	Other users (unknown bias)	Other users (known bias)
What can the attacker time?	Victim's action	Attacker's own action	Attacker's own action
What does the attacker learn?	Victim's input	Unknown bias (typical behavior)	When atypical event happens

Fig. 3: Attack models

deems "hot." In this Java example, the HotSpot 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 the first character of the password (optimized branch) versus when it correctly matches it (non-optimized branch). Figure 2b shows how the side channel disappears when JIT is disabled.

The runtime behavior of the program introduces a side-channel vulnerability, enabling the attacker to learn whether the first character of the guess matches that of the password. This predicate is related to the branch condition in Fig. 1 (right). We present a guide describing how JIT-based side channels can enable an attacker to learn sensitive predicates on input and how to identify potentially vulnerable code. We assume that the attacker is interested in learning a predicate ϕ about any input from a certain subset of all possible inputs to program p. This subset is defined by the assumptions the attacker can make about the input (e.g., that the guess is the same length as the password as in the above example).

A. Attack Models

We now introduce *attack models* that establish the different classes of attacks that we explore and their basic assumptions.

Key to inducing and leveraging JIT-based timing channels is understanding that they arise from a bias in the distribution of inputs to the program. We refer to the act of interacting with a JIT-optimizing runtime in a biased manner as *priming* the runtime. Priming means repeatedly running a program with inputs that exercise certain paths, heating up the state of optimization in a way that favors those paths. A runtime can be primed in various ways, and how we assume it is primed greatly influences what kinds of JIT-based side channels may be used. Another key aspect is time measurement—what exactly do we assume that the attacker is able to time? Lastly, each attack model establishes the purpose of the attack—what does the attacker learn if she succeeds? Figure 3 summarizes the attack models whose details we present in the next sections.

B. Induced-Priming Model

Our first attack model is the *induced-priming model* (IPM), in which we assume that the attacker is able to prime the runtime into a vulnerable state by repeatedly triggering the program p on an input value (or values) of her choice and is then able to time one subsequent call to p made by another user with a secret value s. The attacker's goal is to determine whether s does or does not satisfy some predicate of interest ϕ. This model is only realistic in scenarios where we can assume that the attacker has dominant control of the runtime.

The goal of priming under IPM is to force the runtime into a state where the execution time of the call to p on s is correlated with the value of the predicate ϕ on s. This is done by priming with input values that induce heavier optimization along paths where ϕ is satisfied (or, symmetrically, not satisfied). This results in a "booby-trapped" runtime state in which the timing of a subsequent invocation of $p(s)$ may leak information about the value of $\phi(s)$. Imagine, for example, that there is an ongoing online charity in which participants can donate to one of two political parties. The attacker knows when a particular person will donate and wants to know which party they choose. The attacker can prime the runtime with a flood of small donations to one party, and then time the victim's donation. Its execution time will depend on whether or not the victim's party choice triggers the more optimized program path.

C. Natural-Priming Model

Our second model does not depend on the attacker's ability to control the priming of the runtime. In the *natural-priming model* (NPM), the runtime is primed through a natural bias in the input distribution about predicate ϕ. The attacker can measure the timing of her own call to p (her "probe") on an input of her choice. We study two subcases of this model, differing in what the attacker tries to learn from her probe.

1) Typical Behavior: In the first version (NPM-LTB), the attacker aims to learn the typical behavior of the program. From the timing of her own probe $p(\pi)$, the attacker learns if her input π agrees or disagrees with the typical input to p with respect to the predicate ϕ. Imagine, for example, that there is an online referendum taking place. The attacker wants to know what decision is favored by the majority. If enough users have voted disproportionately in favor of one decision, the runtime could have been primed to favor that choice. The timing of the attacker's probe $p(\pi)$ could thus leak information about whether her vote π represents the typical case.

2) Atypical Behavior: In the second version (NPM-LAB), the attacker aims to learn whether another user's input to the program is atypical with respect to a well-established bias. Again, the attacker uses the timing of her subsequent probe $p(\pi)$ to learn this information. As we will see, depending on which optimizations are involved, a small number of calls or even a single call to p with an atypical value can change the state of the runtime. This may significantly affect the timing of future calls to p. For example, imagine a website where patients of a clinic can obtain their test results for a life-threatening infectious disease. The majority of results come out negative. The attacker can learn when someone tests positive by repeatedly polling her own negative result and watching out for changes in timing.

D. Roadmap and Contributions

While details differ, inferring ϕ under each attack model consists of the same three stages:

1) **Priming:** p is executed repeatedly with an input distribution biased with respect to predicate ϕ.
2) **Timing:** The attacker times the execution of p for a particular input value.
3) **Inference:** Based on the observed execution time, the attacker infers the value of predicate ϕ on unknown input.

Imbalances introduced through biased behavior at runtime are related to various JIT optimizations. These optimizations interact in complex and subtle ways. Combined with noisy timing, this makes the art of leveraging JIT-based timing side channels an intricate process. We identify several *vulnerability templates*, each based on exploitable JIT optimizations. These templates help us identify which predicates related to paths in p may be amenable to JIT-based vulnerabilities, and guide us in finding the right priming parameters or requirements.

While many JIT-based imbalances are small, and thus hard to separate from the noise of a real-world system, we point out that most large JIT-based imbalances consist of many small ones combined. Studying the effects of fine-grained JIT-based vulnerabilities is the initial step toward understanding their contribution to coarse-grained, sizable phenomena.

We first apply our approach in a fine-grained analysis of Java and JavaScript methods. 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 the value of the predicate to quantify how much information about the predicate is leaked. We discuss the results and lessons learned from our most interesting experiments, both successful and unsuccessful.

We then experiment with the Apache Shiro [10] security framework and the GraphHopper [11] route planning server to explore how JIT-based side channels can be induced in large well-known applications. Our results show that they can indeed be introduced, and that they can be sizable enough in magnitude to be observable over the public internet.

Our contributions in this paper are:

- Definition and demonstration of a new class of timing side channels due to JIT optimizations during runtime.
- Three attack models for learning predicates about secret inputs using JIT-based side channels.
- Five vulnerability templates to identify code fragments susceptible to JIT-based timing vulnerabilities.
- A profiling method to gather the statistical information needed to infer predicate values in noisy environments.
- Experimental evaluation of applying our approach to widely used Java and JavaScript functions.
- Examples and experimental analysis of multiple JIT-based side channels in two well-known Java frameworks.

The paper is organized as follows: In Sect. III we review JIT optimizations and introduce related vulnerability templates. In Sect. IV we present algorithms to effectively use timing information arising from JIT-based side channels. In Sect. V we describe our experiments on built-in Java and JavaScript functions. In Sect. VI we discuss the experimental results. In Sect. VII we demonstrate JIT-based side channels in well-known frameworks. In Sect. VIII we discuss related work. In Sect. IX we present our conclusions and ideas for future work.

III. Vulnerability templates for JIT-based side-channels

In this section we review basic characteristics of JIT compilation mechanisms, and then identify vulnerability templates for timing side channels based on JIT compilation techniques.

A. Just-in-time (JIT) compilation

Just-in-time compilation has been used since 1960 [12] to improve the performance of interpreted languages. Based on the observation that a majority of execution time is typically spent executing a small fraction of the code [13], runtime profiling can be used to detect "hot" functions or code portions that are worth feeding into an optimizing compiler. This involves a complex trade-off between initial compilation delays ("warm-up time") and subsequent performance benefits.

Modern JIT compiler implementations involve techniques to dynamically adjust the optimization level (and thus the compilation overhead) of each method in order to maximize the return on investment. Aggressive speculative optimizations may give rise to the need for recurrent assumption checks, and to deoptimization penalties when such a check fails [14].

JIT compilation has been employed by many languages including LISP [15], Smalltalk [16], FORTRAN [17], APL [18], PHP [19], Java [20], and JavaScript [21].

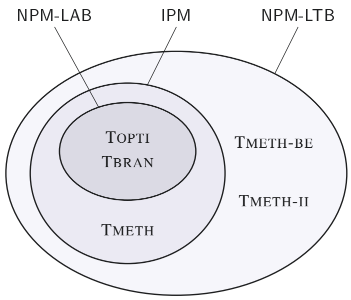


Fig. 4: Vulnerability templates for each attack model.

B. Vulnerability Templates and JIT Compilation Techniques

We show vulnerability templates centered on different JIT compilation techniques. This facilitates identification of code susceptible to a JIT-based side channel and systematic understanding of parameters needed to harness the side channel.

Each vulnerability template has:

- A particular kind of optimization that it exploits.
- A code pattern, e.g., that some method m must be called when ϕ is satisfied and not when ϕ is not satisfied.
- A recipe that guides the search of suitable parameters for priming: how biased the input distribution must be, how many calls to p are needed, etc. This describes the priming the attacker must be able to induce under IPM or the natural priming necessary under NPM.
- The attack model(s) for which the template is harnessable.

As we introduce the templates, we provide the necessary background about the JIT compilation techniques they exploit.

*1) **Branch prediction** (TBRAN):* JIT branch prediction uses counters to track the frequency of each branch of a conditional. When a method is compiled, this information may be used to generate native code where the most taken branch appears first, avoiding a jump instruction. Savings are amplified in the case of loops. This optimization is independent of CPU-level branch prediction, but can achieve positive synergy with it.

Code Pattern: TBRAN can be applied for any predicate directly related to a conditional statement. The imbalance that it introduces is small, so that it may only be observable in specific cases. This template works best in situations where the conditional is enclosed in a loop (which amplifies the small difference), or in small programs, where the small difference achieved is significant w.r.t. the cost of the rest of the program.

Recipe: The amount of priming must be sufficiently high that JIT deems generating the more efficient native code worthwhile. Also, priming must be sufficiently biased so that a high-enough fraction of branching decisions favor one side.

Attack Models: IPM, NPM-LTB, NPM-LAB.

*2) **Optimistic compilation** (TOPTI):* If, when a method is aggressively compiled (e.g. elevated to a higher tier of compilation in HotSpot), the counters shows that one side of a conditional is *very* rare, the optimizing compiler may make the optimistic assumption that the branch will never be taken. Similarly, if counters show that a potentially polymorphic call site always calls the same receivers, the optimizing compiler may assume the rarely-seen dispatches never occur. In either scenario, if and when the rare case occurs and the optimistic assumption is broken, the method must be *de-optimized* and replaced with a slower, more conservative version.

Code Pattern: TOPTI can be applied for any ϕ where satisfying or not satisfying ϕ means that some non-empty branch is never taken or some receiver never called.

Recipe: Priming must ensure that (i) the method containing the conditional is called enough times to be aggressively compiled, and that (ii) by the time that happens, the conditional or dispatch of interest has behaved almost always uniformly.

Attack Models: IPM, NPM-LTB, NPM-LAB.

*3) **Method compilation** (TMETH):* Compilation decisions are impacted by runtime profiling metrics. One key metric is method invocation count. When the number of method invocations reaches a certain threshold [22], the method may be scheduled for compilation or more aggressive recompilation. Another factor that may promote (re)compilation of a method are back-edge counters that track how often backward jumps (typically due to loops) are taken. TMETH exploits the speed difference between interpreted and compiled (or between conservatively and aggressively compiled) code.

Code Pattern: An input satisfying ϕ results in a call to some method m that is not called when ϕ is unsatisfied.

Recipe: Priming must ensure that 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 or augments an observable imbalance in the timing of p.

Attack models: IPM and NPM-LTB. Not exploitable under NPM-LAB, bar extreme conditions. The atypical behavior must impact the runtime state for the attacker to detect it. For TMETH, this means that calls to p on an atypical value impact m's compilation level. To detect this, the attacker's probe needs to execute a path containing m (otherwise the probe's timing would be independent of m's compilation level). But this means that the attacker needs to ensure her own probes are not responsible for the compilation of m, requiring a very nuanced, and generally unrealistic, understanding of the current profile of m.

*4) **Method compilation due to back-edges** (TMETH-BE):* This template, specific to NPM-LTB, exploits method compilation due to back-edge counters rather than method invocation counters. A method no longer needs to be called for one predicate value and not the other. Instead, a method m called the same number of times in both cases is (or is not) compiled (or is compiled to a different level of optimization) depending on whether the back-edge counters are sufficiently high.

Code Pattern: The predicate impacts the number of back edges (jumps to previous code) traversed in a method m.

Recipe: The priming amount must be in the range to induce a difference between the optimization level of m according to the two priming scenarios. The ideal probe value 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.

Attack Models: NPM-LTB.

*5) **Method compilation due to imbalanced invocations** (TMETH-II):* This template is specific to NPM-LTB.

Code Pattern: This template applies to any predicate that impacts the frequency of calls to a method m. The case where m is never called for one predicate value is a specific one.

Recipe: The priming amount must be in the range so that the level of compilation of m is different across the two priming scenarios. The ideal probe value for this case is one in which calls to m are expensive.

Attack Models: NPM-LAB.

IV. Statistical Profiling for Accurate Inference

In this section we discuss how an attacker can use the timing information she collects to correctly infer predicate values. Though not always necessary, the attacker's endeavor can be greatly aided if she builds an informative *profile* of the expected timing distribution under different predicate values.

Two key factors about the predicate ϕ impact the attacker's profiling strategy. First, how many paths through program does the satisfaction of the predicate ϕ (or $\neg\phi$) correspond to? Second, if the predicate corresponds to a set of program paths, are there any additional assumptions the attacker can make about the value of unknown input to p to reduce that set of paths? The more limited this set of program paths is, the simpler it will be to produce a reliable statistical model.

A. Learning under IPM

Under IPM, the attacker primes the runtime engine into a state where the execution time of a subsequent call to p on an unknown value leaks information about whether that value satisfies ϕ. For accurate inference, the attacker can develop a statistical profile of the execution times of p on inputs satisfying ϕ and $\neg\phi$, respectively, after priming with a chosen priming value. The more distinguishable the profiles under the two cases, the more successfully the attacker has booby-trapped the runtime engine.

Obtaining a statistical profile benefits us twofold. First, time measurements are affected by nondeterminism from various sources, from 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 accounts for such noise. Second, the assumption that the attacker has complete control over the runtime is often unrealistic. When we build a statistical profile, we can simulate an environment where some proportion of calls to p is outside the control of the attacker. By *priming with an α distribution* (with respect to ϕ) we mean priming p with inputs satisfying ϕ with probability α, and $\neg\phi$ with probability $1-\alpha$. When we build a statistical profile, we prime with $\alpha < 1$ to simulate a context where p is occasionally triggered on inputs satisfying the opposite value to the one we have chosen to heat up.

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 satisfy any assumptions the attacker makes over the input space, and (b) p_ϕ satisfies ϕ whereas $p_{\neg\phi}$ satisfies $\neg\phi$. The test values t_ϕ and $t_{\neg\phi}$ are chosen randomly from the set of possible secret values (T_ϕ and $T_{\neg\phi}$) satisfying ϕ and $\neg\phi$ respectively (along with any additional assumptions on the input space) to generate representative timing information for a secret value. The priming amount n is the total number of calls to the program p in the Prime algorithm (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 test values can impact the accuracy of the statistical profile. The more similar the test values t_ϕ and $t_{\neg\phi}$ to the actual unknown value, the more accurate the profile likely is. Here similarity means that modulo branch decisions correlated with the predicate, the test input follows a similar program path as the unknown input. Since this cannot be known beforehand, the attacker's best option is to generate and profile for a wide set of test values. Likewise, choice of priming input can impact how successful an attacker is in booby-trapping the runtime. Ideally the attacker would choose priming input following the same program path as the unknown input. If this is not possible, an attacker could vary her priming input over a set of possible paths in an attempt to avoid introducing a timing channel due to an entirely different predicate.

B. Learning under NPM

In NPM, the runtime engine is primed by a natural bias in the input distribution to p. The attacker then executes and times $p(\pi)$ on a probe value π of her choice. The timing is used to infer either a) (NPM-LTB) whether ϕ or $\neg\phi$ is sufficiently dominant among the input to p, or b) (NPM-LAB) if and when a call to p that is atypical with respect to ϕ has been made. As in IPM, we develop a statistical profile to reliably perform inference when presented with the timing of $p(\pi)$. Unlike IPM, the attacker is not in control of the priming and so profiling requires simulating the natural priming of the runtime.

Learning under NPM-LTB: Here the bias of the natural priming is unknown. The attacker instead generates two sets of priming inputs (all values of one set satisfying ϕ and all those of the other satisfying $\neg\phi$) and primes using those values. Again, we introduce the ratio α, this time to simulate differing degrees of bias in the natural priming of p. The attacker generates statistical profiles for the timing of her probe π to p under both possible priming scenarios. The profiling and subsequent inference specific to NPM-LTB is given in Algorithm 3. The accuracy of this statistical profile depends on how closely the set of possible priming input resembles the input actually used to naturally prime the runtime engine.

Learning under NPM-LAB: Here the bias of the priming is known. The attacker can bias in favor of the appropriate value of ϕ using priming values from the corresponding set. She then generates a statistical profile of the timing of her probe π to p after a call has been made to p using a randomly chosen test value t_ϕ. She then does the same for randomly chosen $t_{\neg\phi}$. Here t_ϕ and $t_{\neg\phi}$ are drawn from the set of possible test values satisfying ϕ or $\neg\phi$ as appropriate. The profiling and inference code for NPM-LTB is given in Algorithm 4.

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

Algorithm 1: Prime pseudocode.

```
input : N (profiling amount), n (priming amount), α (ratio),
        x_φ, x_¬φ (priming inputs), T_φ, T_¬φ (profiling test input sets)
v_φ, v_¬φ ← two empty vectors to store timing profiles;
for i ← 1 to N do
    t_φ ← random(T_φ);
    Prime(n, α, x_φ, x_¬φ );
    v_φ.append( Time(p(t_φ)) ) // and start with a fresh runtime
end
for i ← 1 to N do
    t_¬φ ← random(T_¬φ);
    Prime(n, α, x_φ, x_¬φ);
    v_¬φ.append( Time(p(t_¬φ)) ) // and start with a fresh runtime
end
Prime(n, 1.0, x_φ, null);
timingOfSecretInput ← Time(RealCall);
leakageEst ← InferPred(v_φ, v_¬φ, timingOfSecretInput);
```

Algorithm 2: IPM attack pseudocode

```
input : N (profiling amount), n (priming amount), α (ratio),
        X_φ, X_¬φ (profiling priming sets), π (probe)
v_φ, v_¬φ ← two empty vectors to store timing profiles;
for i ← 1 to N do
    x_φ, x_¬φ ← random(X_φ), random(X_¬φ)
    Prime(n, α, x_φ, x_¬φ );
    v_φ.append( Time(p(π)) ) // and start with a fresh runtime
end
for i ← 1 to N do
    x_φ, x_¬φ ← random(X_φ), random(X_¬φ);
    Prime(n, α, x_¬φ, x_φ );
    v_¬φ.append( Time(p(π)) ) // and start with a fresh runtime
end
RealPrime;
timingOfProbeAfterSecretPriming ← Time(p(π));
leakageEst ← InferPred(v_φ, v_¬φ, timingOfProbeAfterSecretPriming);
```

Algorithm 3: NPM-LTB attack pseudocode

```
input : N (profiling amount), n (priming amount), α (ratio),
        X_φ, X_¬φ T_φ, T_¬φ (profiling priming and test sets), π (probe)
v_φ, v_¬φ ← two empty vectors to store timing profiles;
for i ← 1 to N do
    x_φ, x_¬φ ← random(X_φ), random(X_¬φ);
    Prime(n, α, x_φ, x_¬φ );
    call p(t_φ ← random(T_φ)) ;
    v_φ.append( Time(p(π)) ) // and start with a fresh runtime
end
for i ← 1 to N do
    x_φ, x_¬φ ← random(X_φ), random(X_¬φ);
    Prime(n, α, x_φ, x_¬φ );
    call p(t_¬φ ← random(T_¬φ)) ;
    v_¬φ.append( Time(p(π)) ) // and start with a fresh runtime
end
RealPrime;
RealCall;
timingOfProbeAfterSecretBehavior ← Time(p(π));
leakageEst ← InferPred(v_φ, v_¬φ, timingOfProbeAfterSecretBehavior);
```

Algorithm 4: NPM-LAB attack pseudocode

V. Library Evaluation: Setup

We describe our subjects, setup, and decisions. In each case, the program p under test is a method from the standard library.

A. Source of experimental subjects

We evaluated our approach on the **java.math.BigInteger**, **java.lang.Math** and **java.lang.String** classes from the Java standard library (JDK 8, rev. b132) [23] and on the **Math**, **String** and **Array** JavaScript objects from Google's V8 open-source JavaScript engine (V8 4.5.103.35) [24]. We removed methods with no conditionals, native methods not written in Java or JavaScript, duplicates modulo type (e.g., float vs. double), and those with isomorphic control-flow structures. For the remaining methods we applied our approach and chose predicates for conditionals that satisfied the most relevant templates. In the following sections we show a selection of our results featuring the cases (successful and unsuccessful) that we found most interesting and relevant.

B. Computing information leakage via conditional entropy

For each experimental subject we ran 1000 iterations and, on each iteration, we primed the runtime as described in each of the next subsections and timed a 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 value of 0.0 means no leakage, while 1.0 means full leakage of ϕ's value from one timing observation.

C. Using priming distributions to simulate noisy triggering

As discussed in Section IV, the α ratio accounts for the fact that in a realistic scenario, we will not have exclusive control over the state of the runtime engine and the bias under NPM may not be absolute. Table I shows the distributions that we associated with each template under each model for both Java and JavaScript programs.

D. IPM experiments

For each case under IPM, we chose two values for priming: one satisfying ϕ and one satisfying $\neg\phi$, following the approach given in Algorithm 2. We then generated two sets

TABLE I: Priming distributions used in our experiments

	IPM		NPM-LTB		NPM-LAB	
	Java	JS	Java	JS	Java	JS
TOPTI	0.998	1.000	0.998	1.000	0.998	1.000
TMETH	0.950	0.950	0.950	0.950	n/a	n/a
TBRAN	0.900	0.900	0.950	0.950	0.900	0.900
TMETH-BE	n/a	n/a	0.950	0.950	n/a	n/a
TMETH-II	n/a	n/a	0.950	0.950	n/a	n/a

of possible secret inputs satisfying ϕ or $\neg\phi$ respectively and both satisfying a set of additional assumptions over the space of all possible inputs. These assumptions are further discussed in Section B. We primed the runtime engine with the priming values 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 [25] for Java, or the `--trace-opt` option for JavaScript, to determine whether the optimization has taken place. 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 runtime engine using one priming value as described above, then timed a call to the method on a randomly chosen secret value satisfying ϕ. Then we performed another 1000 iterations of the experiment, now timing a call to the method on a randomly chosen secret value satisfying $\neg\phi$. From this data we computed the leakage as explained in V-B.

In addition, we also re-executed all experiments (and leakage computation) 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 runtime was primed more heavily favoring ϕ in the original experiment, it is now primed more heavily with input satisfying $\neg\phi$, and vice versa. This evaluates whether that test subject is reversible.

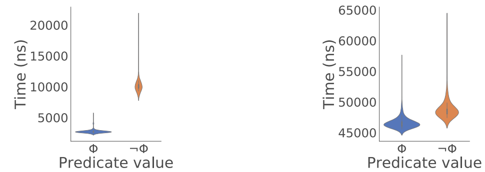
2) Even priming: We re-ran all experiments with a fixed ratio $\alpha = 0.5$, i.e., the amounts of priming satisfying ϕ and $\neg\phi$ are equal. This evaluates the importance of the imbalanced priming ratio in introducing a side channel, as opposed to the more general, overall heating up of the whole 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 very small, fixed, balanced amount of priming (50 calls on both sides) to avoid artificial noise from initial class/method loading delays.

E. NPM experiments

For cases evaluated under NPM, we generated two sets of possible priming values, one satisfying ϕ and the other $\neg\phi$. For each case, we determined the number of priming iterations in the same way as for IPM (see V-D), using JITWatch or `--trace-opt` to guide the search.

NPM-LTB experiments: For evaluation, we repeated the following experiment 1000 times. We primed the runtime engine (with priming parameters obtained as described above) in favor of priming values satisfying ϕ, and then timed a subsequent call on a chosen probe input π. We manually chose π such that the difference between the optimization levels after the two types of priming would be observable. We experimented again by priming the runtime engine with the same parameters as before, but in favor of the priming inputs satisfying $\neg\phi$, and then timed a subsequent call on the same probe π. Each experiment was repeated 1000 times. From this data we computed the leakage as explained in V-B.



(a) Math.max (IPM) (b) Math.max (NPM-LAB)

Fig. 5: Execution time distributions for JavaScript method Math.max under (a) IPM and (b) NPM-LAB.

Since NPM-LTB is the most expressive model in terms of the vulnerability templates applicable under it, we focus our experiments on methods and predicates satisfying templates un-harnessable under the other models.

NPM-LAB experiments: We additionally generated two sets of possible secret inputs satisfying ϕ or $\neg\phi$ respectively and both satisfying a set of additional assumptions over the space of all possible inputs. For evaluation, we did the following 1000 times. We primed the runtime (with priming parameters obtained as above) in favor of the priming values satisfying ϕ, then made a call to the method on a randomly chosen secret value executing the $\neg\phi$ branch. We then timed a subsequent call on a chosen probe input π. Then we performed another 1000 iterations of the experiment, this time calling the method on a randomly chosen secret value executing the ϕ branch and timing the subsequent call on the same probe input π. From this data we computed the leakage as explained in V-B. We then re-ran all experiments under the reverse priming model described in V-D. This evaluates whether the natural priming needs to be more strongly in favor of one particular value of ϕ for atypical behavior to be detected.

F. Hardware setup

The library experiments were run on an Intel NUC 5i5RYH computer (Intel i5-6600K CPU at 3.50 GHz, 32 GB RAM) running Ubuntu Linux 16.04 (kernel 4.4.0-103), the Java 8 Platform Standard Edition version 1.8.0_162 from OpenJDK, and Node.js v4.2.6.

VI. EXPERIMENTAL RESULTS

Tables II, III, and IV summarize our results for Java and JavaScript methods under IPM, NPM-LTB, NPM-LAB respectively. For each set of experiments we report the method name, location of the selected branch instruction in the class source code [23], [24], the template applied, other templates (if any) that also arose unexpectedly, and the priming parameters used. In all cases, we report the amount of information leaked about the predicate value under all evaluated priming scenarios.

A. Optimistic compilation (TOPTI)

When optimistic compilation could be induced, a sizable timing difference arises (e.g., see Fig. 6b), resulting in very reliable learning of $\phi(s)$ under IPM. Our high-leakage results for Java methods BigInteger.min, Math.nextAfter,

TABLE II: Experimental results for IPM in Java (top) and JavaScript (bottom)

Programming language	Method name	Branch instruction	Template (applied, arisen)	Priming amount	Priming ratio (α)	Leakage under α	Leakage under $\overline{\alpha}$	Leakage $0.5 \mid 0.5$	Leakage w/o JIT
Java	BigInteger.min	line 3477	TOPTI	100,000	0.998	**1.00**	**1.00**	0.02	0.06
Java	BigInteger.valueOf	line 1085	TBRAN	10,000	0.900	**0.52**	**0.16**	0.10	0.03
Java	BigInteger.shiftLeft	line 2908	TMETH	10,000	0.950	**0.99**	**0.95**	0.75	0.79
Java	Math.max	line 1316	TBRAN	10,000	0.900	**0.28**	**0.25**	0.04	0.03
Java	Math.ulp	line 1443	TMETH	50,000	0.950	**0.05**	**0.05**	0.02	0.25
Java	Math.nextAfter	line 1926	TOPTI	100,000	0.998	**1.00**	**0.89**	0.02	0.03
Java	Math.min	line 1350	TOPTI	100,000	0.998	**0.03**	**0.01**	0.02	0.03
Java	String.equals	line 976	TOPTI, TBRAN	100,000	0.998	**0.44**	**0.04**	0.12	0.04
Java	String.compareTo	line 1151	TOPTI	100,000	0.998	**0.99**	**0.03**	0.02	0.20
Java	String.startsWith	line 1400	TBRAN	1,000	0.900	**0.46**	**0.16**	0.21	0.25
JavaScript	Math.max	line 73	TOPTI	100,000	1.000	**1.00**	**1.00**	0.07	0.16
JavaScript	Math.trunc	line 176	TBRAN	10,000	0.900	**0.08**	**0.07**	0.08	0.02
JavaScript	Math.aSinh	line 199	TOPTI	100,000	1.000	**1.00**	**0.97**	0.03	0.06
JavaScript	String.charAt	line 71	TOPTI	100,000	1.000	**1.00**	**0.14**	0.07	0.09
JavaScript	String.indexOf	line 118	TBRAN	10,000	0.900	**0.35**	**0.05**	0.15	0.03
JavaScript	Array.toLocaleString	line 224	TOPTI	100,000	1.000	**1.00**	**0.11**	0.06	0.49
JavaScript	Array.slice	line 736	TOPTI	100,000	1.000	**0.04**	**0.04**	0.03	0.04
JavaScript	Array.lastIndexOf	line 1444	TOPTI	100,000	1.000	**0.99**	**0.04**	0.05	0.04

TABLE III: Experimental results for NPM-LTB in Java (top) and JavaScript (bottom)

Programming language	Method name	Branch instruction	Template (applied, arisen)	Priming amount	Priming ratio (α)	Leakage under α
Java	BigInteger.mod	line 2402	TMETH	10,000	0.950	**0.33**
Java	BigInteger.mod	line 2402	TMETH	10,000	0.950	**1.00**
Java	BigInteger.and	line 3054	TMETH-BE	500	0.950	**1.00**
Java	Math.scalb	line 2287	TMETH-BE	5,000	0.950	**0.58**
Java	String.trim	line 2857	TMETH-BE	5,000	0.950	**1.00**
Java	String.replace	line 2060	TMETH-BE, TBRAN	2,000	0.950	**0.92**
Java	String.replace	line 2060	TBRAN	2,000	0.950	**0.66**
Java	String.Constructor	line 250	TMETH-II	500	0.950	**0.08**
JavaScript	Math.min	line 133	TMETH-BE	1,000	0.950	**0.96**
JavaScript	Math.hypot	line 231	TMETH-II	100	0.950	**0.83**
JavaScript	String.split	line 614	TMETH	1,000	0.950	**1.00**
JavaScript	String.concat	line 100	TMETH-BE	1,000	0.950	**0.83**
JavaScript	String.search	line 551	TMETH	1,000	0.950	**0.93**
JavaScript	Array.reverse	line 622	TMETH	100	0.950	**1.00**
JavaScript	Array.map	line 1344	TMETH-BE	1,000	0.950	**0.99**

TABLE IV: Experimental results for NPM-LAB in Java (top) and JavaScript (bottom)

Programming language	Method name	Branch instruction	Template (applied, arisen)	Priming amount	Priming ratio (α)	Leakage under α	Leakage under $\overline{\alpha}$
Java	BigInteger.min	line 3477	TOPTI	100,000	0.998	**1.00**	**1.00**
Java	BigInteger.valueOf	line 1085	TBRAN	10,000	0.900	**0.03**	**0.03**
Java	Math.max	line 1316	TBRAN	10,000	0.900	**0.03**	**0.03**
Java	Math.nextAfter	line 1926	TOPTI	100,000	0.998	**1.00**	**1.00**
Java	Math.min	line 1350	TOPTI	100,000	0.998	**0.04**	**0.03**
Java	String.equals	line 976	TOPTI, TBRAN	100,000	0.998	**0.04**	**0.03**
Java	String.compareTo	line 1151	TOPTI	100,000	0.998	**1.00**	**0.03**
Java	String.startsWith	line 1400	TBRAN	1,000	0.900	**0.05**	**0.03**
JavaScript	Math.max	line 73	TOPTI	100,000	1.000	**0.71**	**0.14**
JavaScript	Math.trunc	line 176	TBRAN	10,000	0.900	**0.04**	**0.03**
JavaScript	Math.aSinh	line 199	TOPTI	100,000	1.000	**0.23**	**0.22**
JavaScript	String.charAt	line 71	TOPTI	100,000	1.000	**0.56**	**0.10**
JavaScript	String.indexOf	line 118	TBRAN	10,000	0.900	**0.04**	**0.03**
JavaScript	Array.toLocaleString	line 224	TOPTI	100,000	1.000	**0.90**	**0.04**
JavaScript	Array.slice	line 736	TOPTI	100,000	1.000	**0.03**	**0.02**
JavaScript	Array.lastIndexOf	line 1444	TOPTI	100,000	1.000	**0.12**	**0.04**
JavaScript	Array.lastIndexOf	line 1444	TOPTI	100,000	1.000	**0.87**	**0.04**

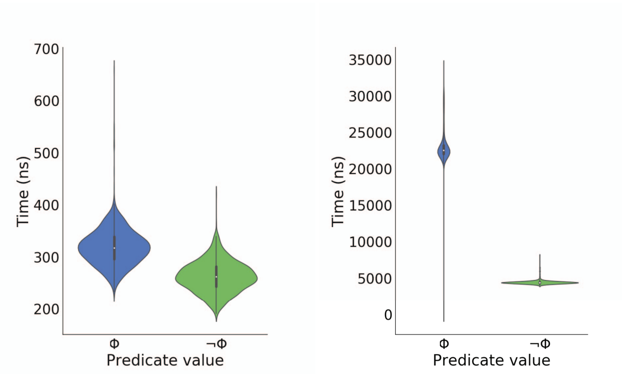
and String.compareTo and JavaScript methods Math.max, Math.aSinh, String.charAt, Array.toLocaleString and Array.lastIndexOf were obtained in this way. Leakage is also reliably high for these methods under NPM-LAB. When the call on the unknown value breaks the optimistic assumption, the runtime engine must revert to a less optimized version of the method under test. This results in an observable difference in the timing of the attacker's probe to the method when compared to the case where the optimistic assumption is not broken (and the highly optimized code used). This timing difference can be augmented by choosing a probe value for which the method is expensive and the difference between versions more apparent. Nevertheless, the magnitude of the timing difference is less than in IPM when the actual execution time of the call breaking the optimistic assumption is measured. Figure 5 demonstrates the relative difference in magnitude across the two attack models for the JavaScript method Math.max.

Both runtime engines require a strong bias for optimistic compilation to occur. For all JavaScript cases, we found that complete bias ($\alpha = 1.00$) is needed to introduce optimistic compilation. In contrast, a very small fraction of input could exercise the uncommon branch before aggressive compilation of the method in the Java cases and optimistic compilation would still occur. Because of this, we used different α distributions for the Java and JavaScript cases exercising TOPTI.

For the other two Java cases, Math.min and String.equals and the JavaScript case Array.slice, our priming did not succeed in inducing optimistic compilation. In Math.min and Array.slice, this was because the functions themselves were not compiled despite their large number of invocations. String.equals was compiled but 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 aggressive 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 p. In contrast, 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 leakage from optimistic compilation due to its less frequent usage elsewhere.

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

There was no notable leakage for the Java methods Math.min and String.equals or the JavaScript method Array.slice under NPM-LAB. This is expected in the cases of Math.min and Array.slice as no optimistic compilation was introduced into the compiled code. For String.equals, there remained the possibility that branch prediction might allow for a timing channel. However, as we will discuss in the section on TBRAN, no such side channel was created. For JavaScript method Math.aSinh, the leakage under NPM-LAB is present, but small. This is because Math.aSinh is inexpensive, making the difference from using its deoptimized version minor.



(a) String.equals (IPM) (b) String.compareTo (IPM)

Fig. 6: Execution time distributions after priming for Java methods (a) String.equals and (b) String.compareTo.

The two results reported for JavaScript method Array.lastIndexOf demonstrate the nuances of choosing a good probe value. Initially, we experimented with a fairly small probe and obtained a leakage of 0.12. Then we tried probing with a more expensive value, obtaining the higher leakage of 0.87, but with a curious side effect that the faster case was actually when optimistic compilation was broken immediately prior to the probe. This means that the deoptimized version of the code was actually faster for our expensive probe. Inspection showed that our expensive probe exercised new behavior and itself triggered deoptimization. When at least some deoptimization had already occurred, this had less of an effect and the execution was faster.

Note that when an attacker succeeds in inducing optimistic compilation, the first call to p that takes the uncommon branch will break the optimistic assumption, and de-optimization will only take place once. Under IPM, this means the attacker must trigger and time p on the secret input before some other user triggers and thus "spoils" the optimistic assumption. Under NPM-LAB, this means that the attacker is only able to observe the first occurrence of atypical program behavior. Once shown false, JIT will not make the same optimistic assumption again.

B. *Method Compilation (*TMETH*)*

Our high-leakage result for BigInteger.shiftLeft exemplifies the potential of TMETH under IPM. In BigInteger.shiftLeft, a different method is called depending on the value of ϕ. With JIT disabled, the execution time *does* leak information about which branch was taken. This is not surprising, as one can expect 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 priming scenarios 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 the Java method Math.ulp, a method is called when input satisfies ϕ but not when it satisfies $\neg\phi$, motivating us to apply the TMETH template. We thought that by compiling that method, we might significantly reduce the execution time on input satisfying ϕ. 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 input satisfying ϕ did not change significantly with its compilation, and did not fall below that of input satisfying $\neg\phi$. 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.

Results for the Java String constructor are similarly explained. Again, we successfully found priming values inducing different levels of optimization in the callee method m but observed minimal leakage. This is due to m performing efficient constant-time computation, making the difference in efficiency between its compiled and un-compiled states indiscernible.

The large majority of function calls made by JavaScript methods in our evaluated libraries were to built-in functions implemented in C. The four JavaScript methods evaluated (Math.hypot, String.split, String.search, and Array.reverse) are some of the few that do call other JavaScript methods large enough to not get inlined. We initially wanted to evaluate these methods under IPM, but in each case, there was a strong side channel even when JIT was disabled (though enabling JIT did allow us to augment these existing side channels). Because of this, we choose to evaluate these methods under NPM-LTB. For each method, we were able to find priming values inducing different optimization levels across the potential secret primings as well as an expensive probe value for which the difference in execution time between the optimized and non-optimized code is sizable.

The difference between the two results for Java function BigInteger.mod stems from an experiment allowing the attacker stronger timing abilities. The second row gives the leakage when we do not time p, but rather its callee m whose compilation we aim to induce. Such refined timing information substantially increases leakage, but requires more assumptions.

C. Branch Prediction (TBRAN)

Branch prediction introduces considerably smaller timing differences than other templates (e.g., see Fig. 6a). Nevertheless, it can still sometimes be exploited to great effect. The Java methods BigInteger.valueOf, String.startsWith, and Math.max and JavaScript method String.indexOf are examples of methods that are small enough that the effect of branch prediction is observable over the computational noise of the method. JavaScript method Math.trunc, on the other hand, is an example where the timing difference is too small to be reliable. Whether or not the branch condition is looped over can also impact the observability of the side channel. In NPM-LTB, where we can choose the test value, 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 iterations of the loop. String.replace (discussed in the next section due to its interaction with TMETH-BE) is an example of this scenario.

Under NPM-LAB, we hoped that branch prediction might be harnessed to detect if atypical behavior occurs. This would occur if executing the method on a secret value that causes the less-seen branch to be taken makes JIT recompile the code to favor the other branch. Differing distributions in the time of the attacker's probe might result. However, our experiments on BigInteger.valueOf, Math.max, String.equals, and String.startsWith and JavaScript method String.indexOf showed this to not be the case. In fact, we even ran experiments were we repeatedly executed the method on test input causing the hitherto less frequent branch to be taken to determine if a heavy change in profiling behavior of the branch would cause JIT to recompile the method. In no cases did this occur, leading us to conclude that TBRAN is not effective under NPM-LAB.

D. Method Compilation via Back Edges (TMETH-BE)

This template is specific to NPM-LTB. Every time we tried to apply TMETH-BE, we successfully found priming amounts such that the method was compiled to different levels of optimization. In the Java cases BigInteger.and and String.trim cases and the JavaScript cases Math.min, String.concat, and Array.map, we were able to find a probe value 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 the Java method Math.scalb, where the maximum possible number of loop iterations is four. The strength of a side channel introduced by TMETH-BE is thus bounded by how expensive the method in question can be made by suitably chosen probe values.

In the Java method String.replace we show an interesting example of interaction between back-edge-induced-compilation and branch-prediction side channels. We again succeed in inducing differing levels of optimization for the same priming amount. But that priming also induced a branch-prediction-based side channel. The timing of $p(t)$ is thus not only affected by the compilation level of the method, but also by branch prediction. Since the priming input satisfying ϕ induced a higher level of compilation, we expected that the timing of the call to $p(t)$ would be faster under that priming. When we choose t to benefit from the branch prediction induced when priming in favor of ϕ, this was the case. This is shown in our first result for String.replace. However, when we choose a probing value that was hindered by the branch prediction induced by priming favoring ϕ, and aided by branch prediction induced when priming favoring $\neg\phi$, the expected outcome was reversed. The timing of $p(t)$ was actually faster under the $\neg\phi$ priming, even though the method had not been compiled because of the unintended branch prediction. This is shown in our second result for String.replace.

VII. Application Leakage Examples

A. Apache Shiro

Apache Shiro [10] is an easy-to-use, open-source Java security framework for authentication. It has over 2000 stars on Github as of this writing. Developers use Shiro to add permissions, roles, and session management to their applications.

1) Shiro Tutorial: The official tutorial shows how to integrate Shiro into your application using a simple user database. Given a username and password, the example code performs a Shiro login with the given credentials, checks them against the database, tests whether the user has a permission, and reports whether they can perform an action. Even this very simple example code could leak under NPM-LAB. Let us imagine that having permission to perform the action, as checked by the if(currentUser.hasPermission(...)) statement in the code, is highly unusual. If an attacker probes the system at regular intervals by timing her own call to the example code, she can find out when someone passes the hasPermission test.

We experimentally demonstrate. Using the unmodified Shiro tutorial code inside a loop, we make unprivileged users prime the system 5000 times by repeatedly logging in and consequently heating up the typical branch; an unprivileged attacker probe the system 200 times through her own login; and a privileged user log in between the attacker's 100'th and 101'st probes. Due to JIT nondeterminism, we repeat the experiment 100 times. Figure 7ab shows the 100 superimposed traces. The point at which the atypical event happens is clearly visible: the attacker's 101'st probe (first one after the event) takes an unprecedented amount of time. The following probes are also more expensive, although the effect soon wears down.

In Figure 7aa we show the null version of the experiment: same conditions, priming, and probing, but the atypical event is replaced with a typical one. This aims at confirming that the phenomenon is caused by the presence of the atypical event, rather than some other aspect of our experimental setup.

We then wrap the same Shiro tutorial program in a simple TCP server. A TCP client connects to the server from a different computer and issues login/action/logout commands. The same priming, probing, and atypical event as before are now executed on the server at the client's requests. Response times are measured on the client side. The LAN setup is described in Section VII-C. Figure 7ac shows that, even partially fuzzed by network noise, the phenomenon is still clearly observable through our LAN. However, it is not strong enough to be realistically observable through the public Internet.

Optimistic compilation is the enabler of this side channel. The large majority of users do not have the special privilege, resulting in highly compiled code with an optimistic assumption. When the privileged user logs on, the assumption is broken, forcing the JVM to fall back to less optimized code. The change in the timing of the probes reflects this. As recompilation happens, the timing of the probes drops.

2) Amplification through computation: The fact that the previous example leaks is remarkable considering that all it does is to check a permission. Observability can be amplified by calling Shiro's hasPermission method from a function that actually computes something—thus increasing the difference between optimized and unoptimized versions of said function. We tested a simple function that converts an array of points from spherical to Cartesian coordinates. Before the computation, the function checks if(currentUser.hasPermission(...)), as indicated by the Shiro documentation. The code (see Appendix 9) is written in a completely symmetric way, in an attempt to avoid any traditional (non-JIT) timing side channel. It is nevertheless affected by the same JIT-based side channel leakage seen in the previous example, now more amplified.

We perform a similar experiment as before. We can use fewer priming iterations (1000) since additional back-edges cause the function to compile earlier. Figure 7ba shows the null experiment. Figure 7bb shows the results when the atypical event occurs after the 100'th probe. Figure 7bc shows that the effects of reverting to less optimized code are observable over the public Internet, and last until the method is recompiled. This increased observability is due to the now larger difference between the highly optimized and less optimized versions of the function. While we experimented with a specific non-trivial, highly-optimizable function, any method involving computation satisfying similar properties and containing a Shiro permission check would be similarly susceptible.

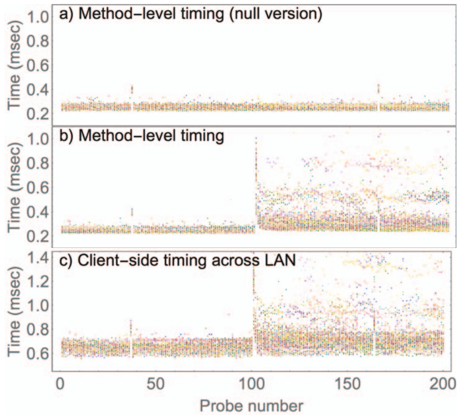
It is noteworthy that, to harness side channels under NPM-LAB, the attacker *only needs to time her own probes*. In contrast to many traditional side channels, the attacker actually infers sensitive information from computation we would expect to be entirely independent from that information. While this increases the applicability of NPM-LAB, the non-resetability of this side channel also deserves note. Once the optimistic assumption is broken, the compiler will not reintroduce the same optimistic compilation again. This means the attacker can detect only the *first* occurrence of the rare event. Nevertheless, for highly sensitive events, this kind of vulnerability is critical. Additionally, if the attacker is able to force the JVM to reset (should she be a system admin of a company or able to force a reset through an orthogonal denial-of-service attack), then she can continue her detection of rare behavior.

B. GraphHopper

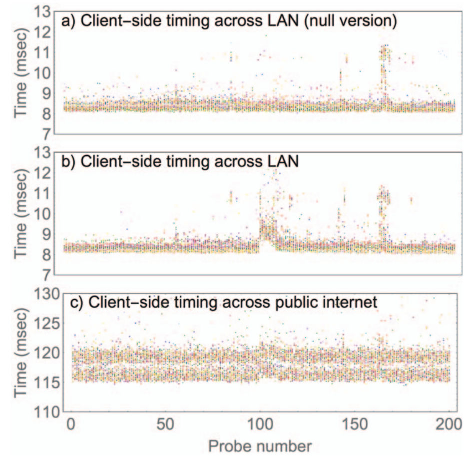
GraphHopper [11] (GH) is an open-source framework that computes directions on city maps. It uses maps from the OpenStreetMap [26] project. The GH server can answer queries like "best route from A to B by train in Berlin" issued by clients through a RESTful API. GraphHopper is a well-known project with over 1,800 stars on Github as of this writing.

We present two examples of leakage due to optimistic compilation in GH. One allows an attacker to discover when someone issues a query in which the origin and destination points are further than a certain threshold apart. The second one allows an attacker to find out when someone issues a query with a certain preference of routing algorithm. Both side channels ultimately adhere to the TOPTI template, though their presence in a large application makes their behavior more intricate. We again evaluate under NPM-LAB since it makes the fewest assumptions about the attacker's capabilities.

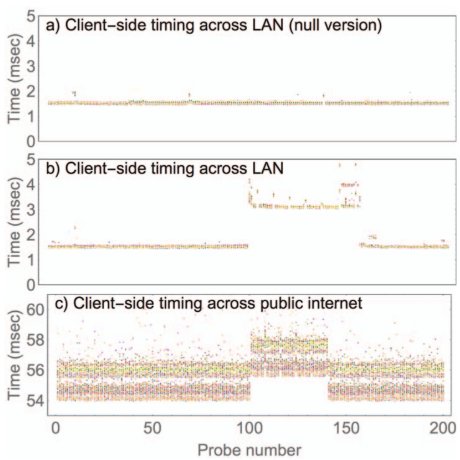
We did not modify GraphHopper in any way. Our experiments can be replicated using the unmodified current distribution of the GH server (VII-C) and the map of Berlin.



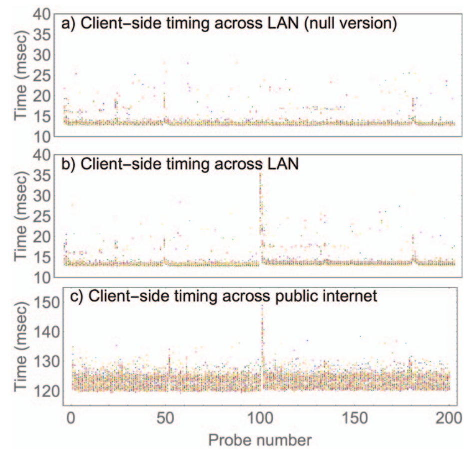
(a) Apache Shiro: Tutorial example (unmodified).



(c) GraphHopper: Maximum distance vulnerability.



(b) Apache Shiro: Tutorial example (augmented).



(d) GraphHopper: Atypical algorithm vulnerability.

Fig. 7: Plots for Apache Shiro and GraphHopper experiments

1) Distance Threshold: GH has a configurable maximum separation (graph edges) between allowable *from* and *to* points. We used a limit of 5000 edges. We experimented under the assumption that the majority of users issued queries within range. We collected 100 traces of the following experiment. We primed the JVM with 3000 routing queries between random locations within range; probed with a routing query between two fixed locations within range; and made a routing query between two random locations outside of the range between the 100'th and 101'st probe. Figure 7cb shows the results over the LAN. Figure 7ca shows the null version. Figure 7cc shows the results over the public Internet. Though not perfectly reliably, this timing channel is observable over the public Internet and the attacker is likely able to infer if and when a user makes an atypical routing query.

2) Routing Algorithm: GraphHopper defaults to Dijkstra's algorithm for routing computation. The API allows the user to select a different one, such as the A* algorithm. If the typical case is Dijkstra, an attacker can probe regularly to detect when another user atypically asks to use the A* algorithm.

We collected 100 traces of the following experiment. We primed the JVM with 1000 routing queries between random locations using Dijkstra's algorithm; probed with a routing query between two fixed locations using Dijkstra's algorithm; and made a routing query on two random locations using the A* algorithm between the 100'th and 101'st probe. Figure 7db shows the results over a LAN. Figure 7da shows the null version. Figure 7dc shows the results across the public Internet. The shift in the timing of the probes is observable over the public Internet. In fact, the probe following the rare behavior takes much longer than any prior probe (usually by ~15 msec) due to the less compiled version of the relevant routing-algorithm-handling code being noticeable more expensive for probes requiring many iterations of the algorithm. However, using such an expensive probe means many back edges are taken, resulting in quick recompilation and a fading effect.

C. Experimental setup

We ran Apache Shiro v1.3.2 and GraphHopper v11.0 on two Intel NUC 5i5RYH computers. Both machines are on our

Ethernet LAN via a Netgear GS108Ev3 switch. Another five computers are on the LAN. Under low load, typical round-trip time between the client and the server machines through the LAN was 0.27 msec (min 0.25, mean 0.273, max 0.34).

For the public Internet experiments we ran the server on the same NUC 5i5RYH computer in our lab, and the client on a remote machine located about 2000 miles away. According to traceroute, the route comprises 10 hops. The remote machine is a shared webserver that hosts 20+ live websites. Round-trip time and noise vary depending on load, but typical RTT was around 55 msec (min 54.1, mean 55.04, max 57.7).

VIII. 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 [27]. 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 each method called within the multiplication code, she can infer the order of the sequence of doublings and additions performed. Page also proposes some potential mitigations at both the language level and the virtual machine level. 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 through bias in input distribution. We *actively* exploit JIT's focus on optimization to create side channels and demonstrate our approach on real applications.

In work complementary to ours, Cleemput et al. [28] propose leveraging the profiling information used in dynamic compilation to mitigate timing side channels. Starting from a developer-chosen root method, profiling information on the number of back edges taken or method invocations is collected for values in a training input set. Based on this process, a set of methods potentially vulnerable to timing channels is selected. Control-flow and data-flow transformations are then applied to reduce their susceptibility to side channels. Control-flow transformations, such as if-conversion, from their paper would aid in protecting sensitive functions from JIT-based side channels. In fact, there is existing work on compiler based strategies for mitigating side-channel vulnerabilities which might be germane to that purpose [29]–[31]. However, none of the solutions they offer have been integrated into the widely used runtimes explored in our work. Frassetto et al. [32] propose JITGuard, a guard for JIT compilers against code-injection, code-reuse and data-only attacks. However, side-channel vulnerabilities are out of scope of their work. Ahead-of-time compilation, where a compilation to native machine code happens *before* execution rather than during, is a potential mitigation for JIT-induced side channels. Since compilation happens before runtime, it is agnostic to input distribution. Ahead-of-time compilation is an option in some versions of Java [33], [34] and JavaScript [35], [36], but generally is not standard or even always available.

Static Side-Channel Analysis: The problem of statically detecting side channels in software has been widely addressed. Antopoulos et al. [37], Chen et al. [38] and Brennan et al. [39] 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 [40] and even synthesizing input so as to maximize the amount of information that can be extracted through the side channel [41], [42]. These approaches rely on a cost model that statically approximates observable information (e.g., 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 runtime. The state of the runtime 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. Some approaches to side-channel analysis include a dynamic component where runtime information is collected and statistical inference performed [43], [44]. However, none consider the space of possible primed runtime environments.

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], [45]–[49] 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 channels in security-related code [50]–[52]. Since then, the CPU's Branch Prediction Unit has been exploited to introduce various flavors of timing channel vulnerabilities [53]–[55].

IX. Conclusions and future work

We presented a new class of runtime-behavior-dependent timing side channels that fundamentally differ from traditional, static-code-dependent side channels. JIT compilation introduces these side channels due to non-uniformity in a program's input distribution with respect to certain predicates. We proposed three attack models under which these side channels are harnessable and five vulnerability templates to detect susceptible code and predicates. We presented a fine-grained analysis of JIT-based side channels on Java and JavaScript functions and demonstrated JIT-based timing channels observable over the public Internet in well-known frameworks.

In the future, we plan to develop a fuzzing strategy over possible priming and test values to induce and evaluate side channels automatically. We also plan to develop an online statistical strategy for detection of atypical behavior under NPM-LAB. We believe that with robust statistical models and enough engineering effort, JIT-based side channels can be used to learn sensitive information in the wild and are worth continued exploration.

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] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[9] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[10] Apache. Apache Shiro. http://shiro.apache.com/.

[11] GraphHopper. GraphHopper. http://www.graphhopper.com/.

[12] J. Aycock, "A brief history of just-in-time," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 97–113, Jun. 2003. [Online]. Available: http://doi.acm.org/10.1145/857076.857077

[13] D. E. Knuth, "An empirical study of fortran programs," *Software: Practice and experience*, vol. 1, no. 2, pp. 105–133, 1971.

[14] 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.

[15] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[16] L. P. Deutsch and A. M. Schiffman, "Efficient implementation of the smalltalk-80 system," in *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1984, pp. 297–302.

[17] G. J. Hansen, "Adaptive systems for the dynamic run-time optimization of programs," Tech. Rep., 1974.

[18] P. S. Abrams, "An apl machine," Tech. Rep., 1970.

[19] J. Watkins. PHP 8 will include a JIT compiler. https://blog.krakjoe.ninja/2019/03/php-gr8.html.

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

[21] Google. TurboFan, an optimizing JIT compiler for the V8 JavaScript engine. https://v8.dev/docs/turbofan.

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

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

[24] V8 source code (Google Git repository), tag b6cb. https://chromium.googlesource.com/v8/v8/+/refs/tags/4.5.103.35/src/.

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

[26] OpenStreetMap. OpenStreetMap. https://www.openstreetmap.org/.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] T. Frassetto, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Jitguard: hardening just-in-time compilers with sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2405–2419.

[33] JEP 295: Ahead-of-Time Compilation. http://openjdk.java.net/jeps/295.

[34] Ahead-of-timeCompilation. https://www.graalvm.org/docs/reference-manual/aot-compilation/#ahead-of-time-compilation.

[35] H. Park, W. Jung, and S.-M. Moon, "Javascript ahead-of-time compilation for embedded web platform," in *2015 13th IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)*. IEEE, 2015, pp. 1–9.

[36] R. Zhuykov, V. Vardanyan, D. Melnik, R. Buchatskiy, and E. Sharygin, "Augmenting javascript jit with ahead-of-time compilation," in *2015 Computer Science and Information Technologies (CSIT)*. IEEE, 2015, pp. 116–120.

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

[38] 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.

[39] T. Brennan, S. Saha, T. Bultan, and C. S. Păsăreanu, "Symbolic path cost analysis for side-channel detection," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 27–37.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

[47] 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.

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

[49] 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.

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

[51] 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.

[52] 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.

[53] 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.
[54] 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.
[55] 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.
[56] Oracle. The Java HotSpot Virtual Machine at a glance. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html.

APPENDIX

A. Other JIT optimizations

We briefly overview other JIT optimization techniques, which, though not directly exploited in any of our vulnerability templates, may nevertheless impact the appearance or strength of a side channel.

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 very nuanced, as it interacts with other optimizations 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 experiments is based solely on inlining, we do use this optimization in combination with other ones (see Section VI). JIT compilation can feature 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 about specific optimizations used in the HotSpot and V8 runtime engines, we refer the reader to the documentation [21], [56].

B. Assumptions on Input Space

We evaluate the IPM and NPM-LAB cases with secret test values that satisfy a certain set of assumptions. In some cases, there were no assumptions made. In the majority of cases, the assumptions 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. This observation also makes more reasonable the NPM-LTB assumption that the values we choose for priming are representative of the set of possible priming values satisfying the assumptions and agreeing on ϕ more reasonable. Even in the few cases where the assumptions were more nuanced, such as in Java method String.replace, they were still very reasonable (the character to be replaced must not be the same as the one it will be replaced by). The only case where we used a stronger assumption about the set of secret values is BigInteger.shiftLeft where the test set of values was constrained in a non-trivial way. When a test value was above a certain threshold, it introduced unexpected behavior into the program. Thus we placed an upper limit on the magnitude of the test values. Nevertheless, we had both positve and negative test values that different by thousands.

```
if (currentUser.isPermitted("seeSecretData")) {
    log.info("Secret data being accessed");
} else {
    log.info("Public data being accessed");
}
```

Fig. 8: Code example of Apache Shiro permissions checking.

```
public static double[] compute(PointPair[] points, Subject currentUser) {
    double x=1, y=1, z=1;
    double a=1, b=1, c=1;
    double[] result = new double[points.length];

    // Use Shiro to check permission as seen in the tutorial.
    if (currentUser.isPermitted("seeSecretData")) {
        log.info("Secret data being accessed");
    } else {
        log.info("Public data being accessed");
    }

    for(int i=0; i<points.length; i++) {
        // Convert first point to rectangular coordinates
        x = points[i].p1.r*Math.sin(points[i].p1.theta)*Math.cos(points[i].p1.phi);
        y = points[i].p1.r*Math.sin(points[i].p1.theta)*Math.sin(points[i].p1.phi);
        z = points[i].p1.r*Math.cos(points[i].p1.theta);

        // Convert second point to rectangular coordinates
        a = points[i].p2.r*Math.sin(points[i].p2.theta)*Math.cos(points[i].p2.phi);
        b = points[i].p2.r*Math.sin(points[i].p2.theta)*Math.sin(points[i].p2.phi);
        c = points[i].p2.r*Math.cos(points[i].p2.theta);

        result[i] = Math.sqrt((x+a)*(x+a)+(y+b)*(y+b)+(z+c)*(z+c));
    }

    return result;
}
```

Fig. 9: Code example from the Apache Shiro Tutorial augmented by performing some computation.