DeJITLeak: Eliminating JIT-Induced Timing Side-Channel Leaks

Qi Qin
qinqi@shanghaitech.edu.cn
ShanghaiTech University
Shanghai, China

JulianAndres JiYang
jlandres@shanghaitech.edu.cn
ShanghaiTech University
Shanghai, China

Fu Song*
songfu@shanghaitech.edu.cn
ShanghaiTech University
Shanghai, China

Taolue Chen
t.chen@bbk.ac.uk
Birkbeck, University of London
London, UK

Xinyu Xing
xinyu.xing@northwestern.edu
Northwestern University
Evanston, Illinois, USA

ABSTRACT

Timing side-channels can be exploited to infer secret information when the execution time of a program is correlated with secrets. Recent work has shown that Just-In-Time (JIT) compilation can introduce new timing side-channels in programs even if they are time-balanced at the source code level. In this paper, we propose a novel approach to eliminate JIT-induced leaks. We first formalise timing side-channel security under JIT compilation via the notion of time-balancing, laying the foundation for reasoning about programs with JIT compilation. We then propose to eliminate JIT-induced leaks via a fine-grained JIT compilation. To this end, we provide an automated approach to generate compilation policies and a novel type system to guarantee its soundness. We develop a tool DeJITLeak for real-world Java and implement the fine-grained JIT compilation in HotSpot JVM. Experimental results show that DeJITLeak can effectively and efficiently eliminate JIT-induced leaks on three widely adopted benchmarks in the setting of side-channel detection.

CCS CONCEPTS

• **Software and its engineering** → **Formal software verification**; • **Theory of computation** → **Program analysis**; • **Security and privacy** → **Formal security models**; **Logic and verification**.

KEYWORDS

JIT compilation, timing side-channel, formal semantics, type inference, detection, mitigation

ACM Reference Format:
Qi Qin, JulianAndres JiYang, Fu Song, Taolue Chen, and Xinyu Xing. 2022. DeJITLeak: Eliminating JIT-Induced Timing Side-Channel Leaks. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549150

*Corresponding author

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11.
https://doi.org/10.1145/3540250.3549150

1 INTRODUCTION

Timing side-channel vulnerabilities in programs arise when the execution time of a program is correlated with secrets, thus pose a serious threat to secure systems. One notorious example is the Lucky 13 attack that can remotely recover plaintext from the CBC-mode encryption in TLS due to an unbalanced branch statement [3].

Constant-time and time-balancing are two programming principles to mitigate timing side-channel vulnerabilities [7]. The former ensures that secrets do not influence control-flow paths, memory access patterns, etc., thus requires significant changes to programs (e.g., complicated bitwise-operations). The latter ensures that each secret branching statement has balanced execution time, and is much easier to achieve in practice. Developing constant-time and time-balanced programs [2] is not easy. Even worse, in practice, they may still be vulnerable if the runtime environment is not fully captured by constant-time or time-balancing models. For instance, static compilation from programs to low-level counterparts can compromise constant-time security [11–13, 25]; constant-time executable programs are vulnerable in modern processors due to, e.g., speculative or out-of-order execution [21, 26, 42, 46]; Just-In-Time (JIT) compilation may undermine time-balanced programs [17, 19].

In this work, we focus on JIT compilation induced leaks (JIT-induced leaks) which could be exploited remotely in real-world applications [17], but currently no rigorous approach can eliminate them other than tuning off JIT compilation [16].

We first lay the foundations for timing side-channel security under JIT compilation by presenting a formal operational semantics. With this, we present the first formalism of timing side-channel security under JIT compilation via the notion of time-balancing. To be generic, we do not model concrete JIT compilation as in [8, 31] which aimed to prove the correctness of JIT compilation. Instead, we leave JIT compilation abstract in our model, which is formalized via compilation directives and allows to consider powerful attackers who have control over JIT compilation. This approach can also enable reasoning about bytecode running with JIT compilation and uncover how code can leak secrets due to JIT compilation in a principled way. We then propose to prevent JIT-induced leaks via a fine-grained JIT compilation and present a type system for statically inferring effective compilation policies.

Based on these results, we present DeJITLeak, a practical tool for generating compilation policies of Java programs that can be proven to completely eliminate JIT-induced leaks, while still benefiting from the performance gains of JIT compilation; in addition,

a lightweight variant of DeJITLeak, DeJITLeak$_{\texttt{light}}$, can eliminate most of the leaks with a low overhead for more performance-conscious applications and is still sound if methods invoked in both sides of each secret branching statement are the same. We also implement the fine-grained JIT compilation in HotSpot JVM from OpenJDK. We conduct extensive experiments on three widely used datasets in recent side-channel detection: Blazer [6], Themis [23], CoCo-Channel [18], DiffFuzz [51], and JVMFuzz [19]. Experimental results show that DeJITLeak significantly outperforms the strategies proposed in [16]. We report interesting case studies which shed light on further research. In summary, our contributions are:

- A formal treatment of JIT-induced leaks including an operational semantics and a time-balancing notion under JIT compilation;
- A protection mechanism against JIT-induced leaks via a fine-grained JIT compilation and an efficient approach to generate JIT compilation policies for fine-grained JIT compilation with security guarantees;
- A practical tool that implements our approach and extensive experiments to demonstrate the efficacy of our approach.

Structure. Section 2 briefly introduces JIT-induced leaks and presents an overview of our approach. Section 3 formalises timing side-channel security under JIT compilation. In Section 4, we propose a protection mechanism and a type system to guarantee its soundness. Section 5 presents an implementation of our approach for real-world Java. Section 6 reports an extensive evaluation. We discuss related work in Section 7 and conclude this work in Section 8.

2 OVERVIEW

In this section, we first give a brief introduction of JIT-induced leaks [17]. We will exemplify these leaks using the HotSpot JVM (HotSpot for short) on OpenJDK 1.8. We then give an overview of our approach to identify and eliminate the JIT-induced leaks.

2.1 JIT-Induced Leaks

JIT-induced leaks could be caused by at least the following three JIT compilation techniques [17].

Optimistic compilation (Topti). Optimistic compilation is a type of speculation optimizations [8]. During the JIT compilation of a method, the compiler speculates on the most likely executed branches by pruning rarely executed branches. As a result, it reduces the amount of time required to compile methods at runtime and space to store the native code. However, there might be a subsequent execution where the speculation fails and the execution must fall back to bytecode in the interpreted mode. To handle this issue, a deoptimization point (a.k.a. uncommon trap) is added to the native code and, when encountered, deoptimization is performed which recovers the program state and resumes execution using bytecode. Clearly, executing the native code after compilation is much more efficient if no deoptimization occurs. However, when deoptimization occurs, it will take longer time to deoptimize and roll back to the bytecode. This difference in execution time induces a timing side-channel even if branches are balanced in bytecode.

As an example, consider the pwdEq method shown in Figure 1a, which is extracted and simplified from the DARPA Space/Time Analysis for Cybersecurity (STAC) engagement program *gabfeed_1* [58]. It takes the strings a and b with length 8 as inputs denoting the user-entered and correct passwords respectively. It checks if the two strings are identical (the for loop). The flag equal is assigned by false if two chars mismatch. To balance execution time, the dummy flag shmequal is introduced.

The pwdEq method is marked as safe in STAC and would be verified as safe by the timing side-channel verification tools Blazer [6] and Themis [23] which do not consider JIT compilation. However, indeed it is vulnerable to Topti. To trigger Topti, we execute pwdEq 50,000 times using two strings "PASSWORD" and "password". After that, the else-branch is replaced by the corresponding uncommon trap, so the costly deoptimization will perform later. To trigger this, we use two strings x and y with length 8 such that $x[0]$ is 'p', $y[0]$ is not 'p', and the rest is the same. We collect the execution time of pwdEq with inputs (x,"password") and (y,"password") respectively. The distribution of the execution time is shown in Figure 1b. In comparison, Figure 1c shows the distribution of execution time with JIT compilation *disabled*. We can observe that the difference in the execution time between two branches is much larger when JIT compilation is enabled, allowing an attacker to infer if the first char is correctly guessed. Our approach prevents this leak by disabling the optimistic compilation optimization of the conditional statement in pwdEq rather than disabling JIT compilation. The effectiveness is justified by the execution time depicted in Figure 1d. It is more efficient than natively disabling JIT compilation (e.g., Figure 1c vs. Figure 1d).

Branch prediction (Tbran). Branch prediction is a conservative optimization of conditional statements. Instead of pruning rarely executed branches, branch prediction generates native code by reordering the basic blocks to avoid jumps over frequently executed branches and thus improves the spatial locality of instruction cache. However, the reordering of basic blocks unbalances the execution time of branches even if it is balanced in bytecode. Although the difference in the execution time between branches via Tbran is small for a single conditional statement, it may be amplified by repeated executions (e.g., enclosed in a loop).

Method compilation (Tmeth). The most fundamental feature of JIT compilation is method compilation, which can be triggered if a method is frequently invoked or some backward jumps are frequently performed. Meanwhile, during compilation frequently invoked small methods could be inlined to speed up execution. If an attacker can enforce some methods in a branch to be frequently invoked in advance so that those methods are (re)compiled or inlined, the execution time of this branch may be shortened. This difference in execution time between branches would induce a timing side-channel.

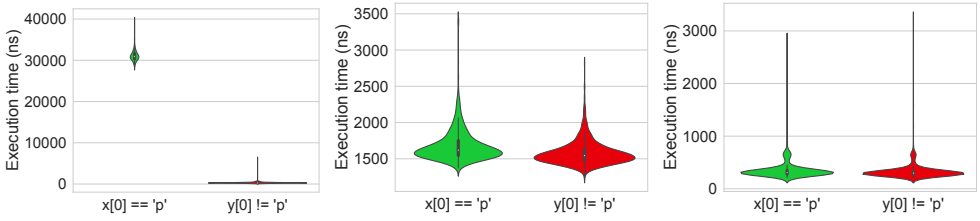
Concrete demonstrations of Tbran and Tmeth refer to [56].

2.2 Eliminating JIT-Induced Leaks

Assuming that a program in bytecode is time-balanced, our goal is to automatically prevent it from the JIT-induced leaks. One possible way is to adopt constant-time programming principle (e.g., [1, 6, 22, 22, 23, 49, 66]). However, there are two limitations: (i) significant changes have to be made (e.g., complicated bitwise-operations), making reasoning about functional-correctness harder. For instance, OpenSSL applied a >500 LOC patch to perform constant-time cipher block chaining (CBC) decoding, the complexity of which led

```
boolean pwdEq(char[] a,char[] b){
    boolean equal = true;
    boolean shmequal = true;
    for (int i = 0; i < 8; i++) {
        if (a[i] != b[i])
            equal = false;
        else shmequal = false;
    }
    return equal; }
```

(a) The pwdEq method



(b) JIT enabled (c) JIT disabled (d) Mitigated

Figure 1: The pwdEq method and its execution time with JIT enabled and disabled under Topti

to subsequent issues [7]. To the best of our knowledge, no tool can automatically rewrite a Java program to a constant-time one. (ii) Different from programs written in static programming languages [22, 66] for which constant-time written is done once for each program, for programs that can be JIT compiled, as compilation may destruct constant-time security [11–13, 25], constant-time security should be enforced during each JIT compilation, incurring large overhead to JIT complication.

Another straightforward way to prevent JIT-induced leaks is to simply disable JIT compilation completely or JIT compilation of the chosen methods. Indeed, [16] proposed three compilation strategies: NOJIT, DisableC2 and MExclude. (i) The NOJIT strategy directly disables JIT compilation (e.g., both the C1 and C2 compilers in HotSpot), so no method will be JIT compiled. This strategy is effective and convenient to deploy, but could lead to significant performance loss. (ii) The DisableC2 strategy only disables the C2 compiler instead of the entire JIT compilation, by which the leaks induced by the C2 compiler (e.g., Topti) can be prevented, but not for Tbran or Tmeth. This strategy also sacrifices the more aggressive C2 optimization and hence may suffer from performance loss. (iii) The MExclude strategy disables JIT compilation for the user-chosen methods instead of the entire program. Its main shortcoming is that non-chosen methods may be still vulnerable, and it is also unclear how to choose methods to disable. In [16], MExclude is applied to the methods that contain secret branches which can prevent Tbran and Topti leaks, but not Tmeth leaks. In summary, these compilation strategies either incur a high performance cost or fail to prevent all the known JIT-induced leaks.

In this work, we first lay the foundations for timing side-channel security under JIT compilation by presenting a formal operational semantics and defining a notion of time-balancing for a fragment of the JVM under JIT compilation. It allows us to reason about timing side-channel security of bytecode programs running with JIT compilation in a principled way. Based on our formalism, we observe that secret information can only be leaked when there is a conditional statement whose condition relies on secret data, and at least one of the following cases occurs, namely,

(1) (Tmeth leaks) a method invoked in a branch is JIT compiled or inlined;
(2) (Tbran leaks) the conditional statement is optimized with the branch prediction optimization;
(3) (Topti leaks) the conditional statement is optimized with the optimistic optimization;

Therefore, disabling JIT compilation at the method level is indeed unnecessary for preventing JIT-induced leaks, instead, we only need to ensure that secret information will not be leaked when the methods are JIT compiled or inlined.

Based on the above observation, we propose a novel approach DeJITLeak to automatically eliminate JIT-induced leaks. To the best of our knowledge, this is the first work to prevent all the above JIT-induced leaks without disabling any compiler in HotSpot, which is in a sharp contrast with the existing compilation strategies [16].

In a nutshell, DeJITLeak automatically locates secret branch points (program points with conditional statements whose conditions rely on secret data) by a flow-, object- and context-sensitive information flow analysis of Java bytecode [62]. The conditional statements at those secret branch points should not be optimized via branch prediction or optimistic compilations. It then extracts all the methods invoked in those conditional statements and identifies those methods that should not be JIT compiled or inlined. Based on these, we put forward a fine-grained JIT compilation and present a type system to prove the soundness of the fine-grained JIT compilation, i.e., a time-balanced program remains time-balanced under our fine-grained JIT compilation if the program is well-typed under our type system. Note that our approach does not guarantee that all the identified branch points or methods are necessary, but the precision of our approach is assured by the advanced information flow analysis and is indeed validated by experiments in Section 6.

Finally, the fine-grained JIT compilation is implemented by modifying HotSpot. Our experimental results show that our approach is significantly more effective than DisableC2 and MExclude, and is significantly more efficient than NOJIT.

3 FORMALISM OF SECURITY

In this section, we present a fragment of JVM and formalize timing side-channel security via the notion of time-balancing.

3.1 The JVM Submachine

We define a fragment JVM_{JIT} of JVM with conditional and unconditional jumps, operations to manipulate the operand stack, and method calls. For the sake of presentation, both bytecode and native code are presented in JVM_{JIT}. Note that our methodology is generic and could be adapted to real instruction sets of bytecode and native code.

Syntax. Let **LVar** (resp. **GVar**) be the finite set of local (resp. global) variables, **Val** be the set of values, **M** be a finite set of methods. A program P comprises a set of methods and each method is a

$$\frac{m[\text{pc}] = \text{push } v}{\langle \text{pc}, m, \rho, \text{os}\rangle \rightsquigarrow \langle \text{pc}+1, m, \rho, v \cdot \text{os}\rangle} \qquad \frac{m[\text{pc}] = \text{pop}}{\langle \text{pc}, m, \rho, v \cdot \text{os}\rangle \rightsquigarrow \langle \text{pc}+1, \rho, \text{os}\rangle} \qquad \frac{m[\text{pc}] = \text{binop } op \quad v = v_1 \; op \; v_2}{\langle \text{pc}, m, \rho, v_1 \cdot v_2 \cdot \text{os}\rangle \rightsquigarrow \langle \text{pc}+1, m, \rho, v \cdot \text{os}\rangle}$$

$$\frac{m[\text{pc}] = \text{ifeq } j \quad \text{pc}' = (v = 0)?j : \text{pc}+1}{\langle \text{pc}, m, \rho, v \cdot \text{os}\rangle \rightsquigarrow \langle \text{pc}', m, \rho, \text{os}\rangle} \qquad \frac{m[\text{pc}] = \text{ifneq } j \quad \text{pc}' = (v \neq 0)?j : \text{pc}+1}{\langle \text{pc}, m, \rho, v \cdot \text{os}\rangle \rightsquigarrow \langle \text{pc}', m, \rho, \text{os}\rangle} \qquad \frac{m[\text{pc}] = \text{swap}}{\langle \text{pc}, m, \rho, v_1 \cdot v_2 \cdot \text{os}\rangle \rightsquigarrow \langle \text{pc}+1, \rho, v_2 \cdot v_1 \cdot \text{os}\rangle}$$

$$\frac{m[\text{pc}] = \text{store } x \qquad x \in \text{dom}(\rho)}{\langle \text{pc}, m, \rho, v \cdot \text{os}\rangle \rightsquigarrow \langle \text{pc}+1, m, \rho[x \mapsto v], \text{os}\rangle} \qquad \frac{m[\text{pc}] = \text{load } x}{\langle \text{pc}, m, \rho, \text{os}\rangle \rightsquigarrow \langle \text{pc}+1, m, \rho, \rho(x) \cdot \text{os}\rangle} \qquad \frac{m[\text{pc}] = \text{goto } j}{\langle \text{pc}, m, \rho, \text{os}\rangle \rightsquigarrow \langle j, m, \rho, \text{os}\rangle}$$

$$\frac{s \rightsquigarrow s'}{(\text{ch}, \text{h}, s, \text{cs}) \rightarrow (\text{ch}, \text{h}, s', \text{cs})} \qquad \frac{m[\text{pc}] = \text{put } y \qquad y \in \text{dom}(\rho) \qquad s = \langle \text{pc}+1, m, \rho, \text{os}\rangle}{(\text{ch}, \text{h}, \langle \text{pc}, m, \rho, v \cdot \text{os}\rangle, \text{cs}) \rightarrow (\text{ch}, \text{h}[y \mapsto v], s, \text{cs})} \qquad \frac{m[\text{pc}] = \text{get } y \qquad s = \langle \text{pc}+1, m, \rho, \text{h}(y) \cdot \text{os}\rangle}{(\text{ch}, \text{h}, \langle \text{pc}, m, \rho, \text{os}\rangle, \text{cs}) \rightarrow (\text{ch}, \text{h}, s, \text{cs})}$$

$$\frac{m[\text{pc}] = \text{return} \qquad s = \langle \text{pc}', m', \rho', v \cdot \text{os}'\rangle}{(\text{ch}, \text{h}, \langle \text{pc}, m, \rho, v \cdot \text{os}\rangle, \langle \text{pc}', m', \rho', \text{os}'\rangle \cdot \text{cs}) \rightarrow (\text{ch}, \text{h}, s, \text{cs})} \qquad \frac{m[\text{pc}] = \text{deopt md} \quad \mathcal{V}_m > 0 \quad O((\text{ch}, \text{h}, \langle \text{pc}, m, \rho, \text{os}\rangle, \text{cs}), \text{md}) = (\text{h}', s, \text{cs}')}{(\text{ch}, \text{h}, \langle \text{pc}, m, \rho, \text{os}\rangle, \text{cs}) \rightarrow (\text{ch}[m \mapsto \text{base\_version}(m)], \text{h}', s, \text{cs}' \cdot \text{cs})}$$

$$\frac{m[\text{pc}] = \text{return}}{(\text{ch}, \text{h}, \langle \text{pc}, m, \rho, v \cdot \text{os}\rangle, \epsilon) \rightarrow (\text{h}, v)} \qquad \frac{m[\text{pc}] = \text{invoke } m' \quad \text{argv}(m') = x_0, \cdots, x_k \quad \mathbf{d} = \mathbf{d}_\emptyset \quad s = \langle 0, \text{ch}(m'), [x_0 \mapsto v_0, \cdots, x_k \mapsto v_k], \epsilon\rangle}{(\text{ch}, \text{h}, \langle \text{pc}, m, \rho, v_k \cdot \cdots \cdot v_0 \cdot \text{os}\rangle, \text{cs}) \rightarrow_{\mathbf{d}} (\text{ch}, \text{h}, s, \langle \text{pc}+1, m, \rho, \text{os}\rangle \cdot \text{cs})}$$

$$\frac{m[\text{pc}] = \text{invoke } m' \qquad \text{argv}(m') = x_0, \cdots, x_k \qquad \mathbf{d} \in \mathbf{D}_m \qquad \mathbf{d} \neq \mathbf{d}_\emptyset \qquad m'' = \mathbf{d}(m') \qquad \mathcal{V}_{m''} > \mathcal{V}_{m'}}{(\text{ch}, \text{h}, \langle \text{pc}, m, \rho, v_k \cdot \cdots \cdot v_0 \cdot \text{os}\rangle, \text{cs}) \rightarrow_{\mathbf{d}} (\text{ch}[m' \mapsto m''], \text{h}, \langle 0, m'', [x_0 \mapsto v_0, \cdots, x_k \mapsto v_k], \epsilon\rangle, \langle \text{pc}+1, m, \rho, \text{os}\rangle \cdot \text{cs})}$$

Figure 2: Operational semantics of JVM_{JIT}, where $\text{dom}(\rho)$ denotes the domain of the partial function ρ

inst	::=	binop *op*	binary operation on the operand stack
	|	push v	push value v on top of the operand stack
	|	pop	pop value from top of the operand stack
	|	swap	swap the top two operand stack values
	|	load x	load value of x onto the operand stack
	|	store x	pop and store top of the operand stack in x
	|	get y	load value of y onto the operand stack
	|	put y	pop and store top of the operand stack in y
	|	ifeq j	conditional jump
	|	ifneq j	conditional jump
	|	goto j	unconditional jump
	|	invoke m	invoke the method $m \in \mathbf{M}$
	|	return	return the top value of the operand stack
	|	deopt md	deoptimize with meta data md

Figure 3: Instruction set of JVM_{JIT}, where $x \in \mathbf{LVar}$ is a local variable and $y \in \mathbf{GVar}$ is a global variable

list of instructions taken from the instruction set in Figure 3. All these instructions are standard except for deopt md which models uncommon traps (cf. Section 2).

For each method m, $m[i]$ denotes the instruction at the program point i and $\text{argv}(m)$ denotes the formal arguments of m. When a method is invoked, the execution starts with the first instruction $m[0]$. We also denote by $m[i, j]$ for $j \geq i$ the sequence of instructions $m[i]m[i+1]\cdots m[j]$.

Compilation directive. To model method compilation with procedure inline, branch prediction and optimistic compilation optimizations, we use (compilation) directives which specify how the method should be (re)compiled and optimized at runtime. Let $\mathbf{D}_m$ be the set of directives of the method m, and $\mathbf{d}(m)$ the method after JIT compilation according to the directive $\mathbf{d}$. In particular, we use $\mathbf{d}_\emptyset \in \mathbf{D}_m$ to denote no (re)compilation. The formal definition of directives is given in the following subsection.

In general, a method in bytecode is compiled into native code which may be iteratively recompiled later. Hence we assign to each method m a version number $\mathcal{V}_m$, where the bytecode has the version number 0, and $\mathcal{V}_{\max} > 0$ is the highest version number. A directive $\mathbf{d} \in \mathbf{D}_m$ is valid if $m' = \mathbf{d}(m)$ and $\mathcal{V}_{m'} > \mathcal{V}_m$, otherwise $\mathbf{d}$ is an invalid directive. Intuitively, the version number $\mathcal{V}_m$ indicates the optimized level of the method m. JIT recompilation only uses increasingly aggressive optimization techniques, and rolls back to the bytecode version otherwise.

State. A *state* is a tuple $\langle \text{pc}, m, \rho, \text{os}\rangle$, where $\text{pc} \in \mathbb{N}$ is the program counter pointing to the next instruction, $m \in \mathbf{M}$ is the current executing method, $\rho : \mathbf{LVar} \rightarrow \mathbf{Val}$ is a partial function from local variables to values, and $\text{os} \in \mathbf{Val}^*$ is the operand stack. We denote by **States** the set of states. For each function $f : X \rightarrow V$, variable $x \in X$ and value $v \in V$, let $f[x \mapsto v]$ be the function where $f[x \mapsto v](x') = f(x')$ if $x' \neq x$, and $f[x \mapsto v](x') = v$ otherwise. For two operand stacks $\text{os}_1, \text{os}_2 \in \mathbf{Val}^*$, let $\text{os}_1 \cdot \text{os}_2$ denote their concatenation. The empty operand stack is denoted by ϵ.

Configuration. A *configuration* is of the form $(\text{ch}, \text{h}, s, \text{cs})$ or (h, v), where ch is a code heap storing the latest version of methods; $\text{h} : \mathbf{GVar} \rightarrow \mathbf{Val}$ is a (data) heap, i.e., a partial function from global variables to values; $s \in \mathbf{States}$ is the current state; $\text{cs} \in \mathbf{States}^*$ is the call stack, and $v \in \mathbf{Val}$ is a value. Configurations of the form (h, v) are final configurations, reached after the return of the entry point. A configuration $(\text{ch}, \text{h}, \langle \text{pc}, m, \rho, \text{os}\rangle, \text{cs})$ is *initial* if $\text{pc} = 0$, m is the entry point of the program, and $\text{os} = \text{cs} = \epsilon$. **Conf** denotes the set of configurations; $\text{cs}_1 \cdot \text{cs}_2$ denotes the concatenation of two call stacks cs_1 and cs_2; ϵ denotes the empty call stack.

Operational semantics. The small-step operational semantics of JVM_{JIT} is given in Figure 2 as a relation $\rightarrow \subseteq \mathbf{Conf} \times \mathbf{Conf}$, where $\rightsquigarrow \subseteq \mathbf{States} \times \mathbf{States}$ is an auxiliary relation. Note that the directive $\mathbf{d}$ applies to method invocations only.

Instruction push v, pushes the value v on top of the operand stack. Instruction pop, just pops the top of the operand stack. Instruction binop *op* pops the top two operands from the operand stack and pushes the result of the binary operation *op* using these operands. Instruction ifeq j (resp. ifneq j) pops the top v of the operand stack and transfers of control to the program point j if $v = 0$ (resp. $v \neq 0$), otherwise to the next instruction, i.e., the program point $j + 1$. Instruction swap, swaps the top two values of the operand stack. Instruction store x (resp. put y) pops the top of the operand stack and stores it in the local variable x (resp. global variable y). Instruction load x (resp. get y) pushes the value of the local variable x (resp. global variable y), on top of the operand stack. Instruction goto j unconditionally jumps to program point j.

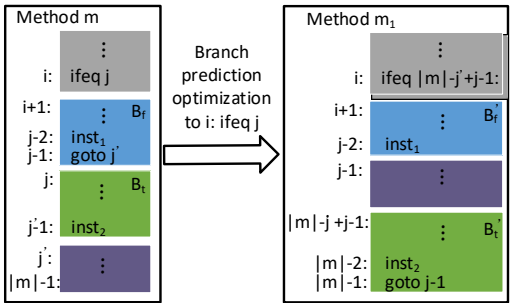


Figure 4: Branch prediction optimization

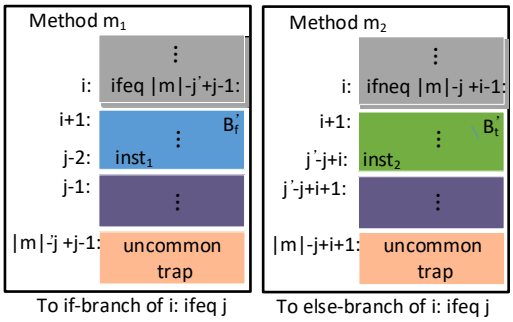


Figure 5: Optimistic compilation optimization

Instruction return ends the execution of the current method and returns the top value v of the current operand stack. If the current method is not the entry point, v is pushed as the top of the operand stack of the caller and the caller is resumed from the return site; otherwise, the final configuration (h, v) is reached. We assume that each method has a unique return instruction which does not appear in conditional statements, as early return often introduces timing side-channel leaks even without JIT compilation.

Instruction deopt md deoptimizes the current executing method and rolls back to the bytecode in the interpreted mode. This instruction is only used in native code and inserted by JIT compilers. Our semantics does not directly model a deoptimization implementation. Instead, we assume there is a deoptimization oracle O which takes the current configuration and the meta data md as inputs, and reconstructs the configuration (i.e., heap h', state s and the call stack cs'). Furthermore, the bytecode version base_version(m) of the method m is restored into the code heap ch. The oracle O results in the same heap h', state s and call stack $\mathsf{cs}' \cdot \mathsf{cs}$ as if the method m were not JIT compiled.

The semantics of invoke m' depends on the directive $\mathbf{d}$. If $\mathbf{d} = \mathbf{d}_\emptyset$, the version of m' in the code heap ch remains the same. If $\mathbf{d}$ is valid, i.e., the version number $\mathcal{V}_{m''}$ of the optimized version $m'' = \mathbf{d}(m')$ is larger than that of the current one $\mathcal{V}_{m'}$, m'' is stored in the code heap ch. After that, it pops up the top $|\mathsf{argv}(m')|$ values from the current operand stack, passes them as the formal arguments to m'', pushes the calling context on top of the call stack and starts to execute m'' in the code heap.

To define a JIT-execution, we introduce the notion of schedules. A *valid schedule* $\mathbf{d}^\star$ for a configuration c is a sequence of valid directives such that the program will not get stuck when starting from c and following $\mathbf{d}^\star$ for method invocations. A valid schedule $\mathbf{d}^\star$ yields a *JIT-execution* $c_0 \Downarrow_{\mathbf{d}^\star} c_n$ that is a sequence $c_0 c_1 \cdots c_n$ of configurations such that c_0 is an initial configuration, c_n is the final configuration, and for every $0 \le i < n$, either $c_i \longrightarrow c_{i+1}$ or $c_i \rightarrow_{\mathbf{d}_i} c_{i+1}$. We require that $\mathbf{d}^\star$ is equal to the sequence of directives along the JIT-execution, i.e., the concatenation of $\mathbf{d}_i$'s. A *JIT-free execution* is thus a JIT-execution $c_0 \Downarrow_{\mathbf{d}_\emptyset^\star} c_n$. In this work, we only consider programs that always terminate.

3.2 JIT Optimization of JVM$_{\mathsf{JIT}}$

We first define branch prediction and optimistic compilation, then define method compilation as well as compilation directives.

Branch prediction. Fix a method m and an instruction $m[i] =$ ifeq j. (ifneq is handled accordingly.) Let B_t (resp. B_f) be the instructions appearing in the if-(resp. else-)branch of $m[i]$, and the last instruction $m[i']$ of B_f is goto j'. The first and last instructions of B_f are $m[i+1]$ and $m[j-1]$ respectively.

If the profiling data show that the program favors the else-branch, the branch prediction optimization transforms the method m into a new method m_1 for $m[i] =$ ifeq j (cf. Figure 4). The formal definition and an illustrating example are given in the technical report [56].

If the profiling data show that the program favors the if-branch, the branch prediction optimization transforms the method m into a new method m_2, similar to m_1, except that (1) the conditional instruction ifeq j is replaced by ifneq $|m| - j + i - 1$ which is immediately followed by the if-branch B_t; (2) the else-branch B_f is moved to the end of the method starting at the point $|m| - j + i - 1$ and the target point of the last instruction goto j' is revised to $j' - j + i + 1$.

We denote by $\mathsf{T}_{\mathsf{bp}}(m, i, \text{else-b})$ and $\mathsf{T}_{\mathsf{bp}}(m, i, \text{if-b})$ the methods m_1 and m_2 respectively. Obviously, the branch prediction optimization transforms the original program to a semantically equivalent one.

Optimistic compilation. Again, consider the conditional instruction $m[i] =$ ifeq j with the if-branch B_t and else-branch B_f. (ifneq can be dealt with accordingly.)

If the profiling data show that the if-branch rarely gets executed, the optimistic compilation optimization transforms the method m into a new method m_1 in a similar way to $\mathsf{T}_{\mathsf{bp}}(m, i, \text{else-b})$ except that the if-branch B_t is replaced by an uncommon trap, as shown in Figure 5 (left-part). The method m_2 is defined similarly if the else-branch rarely gets executed, as shown in Figure 5 (right-part).

We denote by $\mathsf{T}_{\mathsf{oc}}(m, i, \text{else-b})$ and $\mathsf{T}_{\mathsf{oc}}(m, i, \text{if-b})$ the new methods m_1 and m_2 after transformation. It is easy to see that the optimistic compilation optimization is an equivalent program transformation under the inputs that does not trigger any uncommon traps.

Method compilation. At runtime, frequently executed, small methods may be inlined to reduce the time required for method invocations. After that, both branch prediction and optimistic compilation optimizations could be performed. Thus, a compilation directive of a method should take into account procedure inline, branch prediction and optimistic compilation optimizations.

We define a compilation directive $\mathbf{d}$ of a method m as a pair (t, ω), where t is a labeled tree specifying the method invocations

to be inlined, and ω is a sequence specifying the optimizations of branches. Formally, the labeled tree t is a tuple (V, E, L), where V is a finite set of nodes such that each node $n \in V$ is labeled by a method $L(n)$ and the root is labeled by m; E is a set of edges of the form (n_1, i, n_2) denoting that the method $L(n_2)$ is invoked at the call site i of the method $L(n_1)$. We denote by $t(m)$ the new method obtained from m by iteratively inlining method invocations in t. We assume the operand stack of each inlined method is balanced, otherwise the additional pop instructions are inserted. The sequence ω is of the form $(\mathsf{T}_1, i_1, b_1), \cdots, (\mathsf{T}_k, i_k, b_k)$, where for every $1 \le j \le k$, $\mathsf{T}_j \in \{\mathsf{T}_{\mathsf{bp}}, \mathsf{T}_{\mathsf{oc}}\}$ denotes the optimization to be applied to the branch point i_j in the method $t(m)$ with the branch preference b_j. We assume that an index i_j occurs at most once in ω, as at most one optimization can be applied to one branch point.

3.3 Consistency and Time-Balancing

As usual, we assume that each program is annotated with a set of public input variables, while the other inputs are regarded as secret input variables. We denote by $c_0 \simeq_{\mathsf{pub}} c_0'$ if two configurations c_0 and c_0' agree on the public input variables, and denote by $c_0 \simeq_{\mathsf{ch}} c_0'$ if c_0 and c_0' have the same code heap.

Consistency. It is easy to deduce the following theorem which ensures the equivalence of the final memory store and return value from the JIT-free execution and JIT-execution.

Theorem 3.1. *For each initial configuration c_0 of the program P and each valid schedule $\mathbf{d}^\star$ for c_0, we have: $c_0 \Downarrow_{\mathbf{d}^\star_\emptyset} c$ iff $c_0 \Downarrow_{\mathbf{d}^\star} c'$.*

If the output variables are partitioned into public and secret, we denote by $c \simeq_{\mathsf{pub}} c'$ that two configurations c and c' agree on the public output variables.

Theorem 3.2. *For each pair of initial configurations (c_0, c_0') of the program P with $c_0 \simeq_{\mathsf{pub}} c_0'$ and each pair of valid schedules $\mathbf{d}_1^\star$ and $\mathbf{d}_2^\star$ for c_0 and c_0' respectively, we have: $c_0 \Downarrow c$, $c_0' \Downarrow c'$ and $c \simeq_{\mathsf{pub}} c'$ iff $c_0 \Downarrow_{\mathbf{d}_1^\star} c$, $c_0' \Downarrow_{\mathbf{d}_2^\star} c'$ and $c \simeq_{\mathsf{pub}} c'$.*

The theorem states that observing public output variables cannot distinguish secret inputs without JIT compilation iff observing public output variables cannot distinguish secret inputs with JIT compilation.

Time-balancing. To model execution time, we define cost functions for bytecode and native code. Let $\mathsf{cf}_{\mathsf{bc}}$ and $\mathsf{cf}_{\mathsf{nc}}$ be the cost functions for instructions from the bytecode and native code, respectively. We denote by $\mathsf{cf}(\mathsf{inst})$ the cost of the instruction inst, which is $\mathsf{cf}_{\mathsf{bc}}(\mathsf{inst})$ if it is running in bytecode mode, otherwise $\mathsf{cf}_{\mathsf{nc}}(\mathsf{inst})$. We lift the function cf to states and configurations as usual, e.g., $\mathsf{cf}(\langle \mathsf{pc}, m, \rho, \mathsf{os}\rangle) = \mathsf{cf}(m[\mathsf{pc}])$. The cost $\mathsf{cf}(c_0 \Downarrow_{\mathbf{d}^\star} c_n)$ of a JIT-execution $c_0 \Downarrow_{\mathbf{d}^\star} c_n$ is the sum of all the costs of the executed instructions, i.e., $\sum_{i=0}^{n-1} \mathsf{cf}(c_i)$.

Definition 3.3. A program P is *time-balanced* (without JIT compilation) if for each pair of initial configurations (c_0, c_0') of P such that $c_0 \simeq_{\mathsf{pub}} c_0'$ and the code heaps of c_0 and c_0' have the same bytecode instructions, we have: $\mathsf{cf}(c_0 \Downarrow_{\mathbf{d}^\star_\emptyset} c) = \mathsf{cf}(c_0' \Downarrow_{\mathbf{d}^\star_\emptyset} c')$.

Intuitively, the time-balancing requires that two JIT-free executions have the same cost if their public inputs are the same and code heaps have the same bytecode instructions, thus preventing timing side-channel leaks when JIT compilation is disabled.

JIT-time-balancing. To define time-balancing under JIT compilation, called JIT-time-balancing, we first introduce some notations.

Consider a JIT-execution $c_0 \Downarrow_{\mathbf{d}^\star} c_n$ and a method m. We denote by $\mathsf{proj}_m(c_0 \Downarrow_{\mathbf{d}^\star} c_n)$ the projection of the sequence of executed instructions in $c_0 \Downarrow_{\mathbf{d}^\star} c_n$ onto the pairs (i, m') each of which consists of a program point i and a version m' of the method m. A proper prefix π of $\mathsf{proj}_m(c_0 \Downarrow_{\mathbf{d}^\star} c_n)$ can be seen as the profiling data of the method m after executing these instructions, which determines a unique compilation directive of the method m after executed π. We leave runtime profiling abstract in order to model a large variety of JIT compilations and use $\widehat{\pi}$ to denote the profiling data of m after executed instructions π of m or its compiled versions.

Fix a profiler pf that provides one compilation directive $\mathsf{pf}_m(\widehat{\pi})$ of a method m using the profiling data $\widehat{\pi}$. The schedule $\mathbf{d}^\star$ is called a *pf-schedule* if, for each method m and proper prefix π of $\mathsf{proj}_m(c_0 \Downarrow_{\mathbf{d}^\star} c_n)$, the next compilation directive of m in $\mathbf{d}^\star$ after π is $\mathsf{pf}_m(\widehat{\pi})$.

Lemma 3.4. *For each pair of initial configurations (c_0, c_0') of P with $c_0 \simeq_{\mathsf{ch}} c_0'$, and each pair of valid pf-schedules $\mathbf{d}_1^\star$ and $\mathbf{d}_2^\star$ for c_0 and c_0' respectively, we have: for every method m, every pair (π_1, π_2) of proper prefixes of $proj_m(c_0 \Downarrow_{\mathbf{d}_1^\star} c)$ and $proj_m(c_0' \Downarrow_{\mathbf{d}_2^\star} c')$ respectively, if $\pi_1 = \pi_2$ then $pf_m(\widehat{\pi_1}) = pf_m(\widehat{\pi_2})$.*

The lemma ensures that the compilation directives of each method in JIT-executions are the same under the same profiling data.

We assume that, for each time-balanced branching statement in bytecode, the corresponding branching statement in native code is still time-balanced if no JIT optimization is applied. This assumption is reasonable in practice, as both sides of a time-balanced branching statement in bytecode tend to have similar functionality and instruction sequences, thus, time-balanced branches are often nearly balanced after compilation if no JIT optimization is applied. Remark that our formalism is general in principle, as suitable cost functions could be adopted if one prefers to model them precisely.

Definition 3.5. A program P is *JIT-time-balanced* if, for every pair of initial configurations (c_0, c_0') of P with $c_0 \simeq_{\mathsf{pub}} c_0'$ and $c_0 \simeq_{\mathsf{ch}} c_0'$, every pair of valid pf-schedules $\mathbf{d}_1^\star$ and $\mathbf{d}_2^\star$ for c_0 and c_0' respectively satisfies $\mathsf{cf}(c_0 \Downarrow_{\mathbf{d}_1^\star} c) = \mathsf{cf}(c_0' \Downarrow_{\mathbf{d}_2^\star} c')$.

Intuitively, the JIT-time-balancing ensures that two JIT-executions have the same cost if their public inputs and initial code heap are the same and the valid schedules have the same profiler pf for JIT compilation, so it prevents JIT-induced leaks even if the JIT compilation is enabled. Remark that our definition considers powerful attackers who can control executing instructions with chosen inputs before launching attacks so that the code heaps in c_0 and c_0' may be mixed with bytecode and native code and compilation directives are controlled during attacking, which is common in the study of detection and mitigation. In practice, the feasibility of compilation directives depends on various parameters in VM, e.g., whether a method invocation should be inlined depends on its code size, invocation frequency, method modifier, etc.

4 PROTECT MECHANISM AND TYPE SYSTEM

In this section, based on the above formalism, we first propose a two-level protection mechanism to prevent JIT-induced leaks and

$$\frac{m[i] = \text{push } v \quad \mathsf{st}' = \mathsf{pt} \cdot \mathsf{st}}{m, i \vdash (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \mathsf{st}) \Rightarrow (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \mathsf{st}')}\text{T-Push} \qquad \frac{m[i] = \text{binop } op \quad \mathsf{st}' = (\tau_1 \sqcup \tau_2 \sqcup \mathsf{pt}) \cdot \mathsf{st}}{m, i \vdash (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \tau_1 \cdot \tau_2 \cdot \mathsf{st}) \Rightarrow (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \mathsf{st}')}\text{T-Bop} \qquad \frac{m[i] = \text{store } x \quad \mathsf{lt}' = \mathsf{lt}[x \mapsto \tau \sqcup \mathsf{pt}]}{m, i \vdash (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \tau \cdot \mathsf{st}) \Rightarrow (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}', \mathsf{st})}\text{T-Str}$$

$$\frac{m[i] = \text{pop} \quad \mathsf{st} = \tau \cdot \mathsf{st}'}{m, i \vdash (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \mathsf{st}) \Rightarrow (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \mathsf{st}')}\text{T-Pop} \qquad \frac{m[i] = \text{swap} \quad \tau_1' = \tau_1 \sqcup \mathsf{pt} \quad \tau_2' = \tau_2 \sqcup \mathsf{pt}}{m, i \vdash (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \tau_1 \cdot \tau_2 \cdot \mathsf{st}) \Rightarrow (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \tau_2' \cdot \tau_1' \cdot \mathsf{st})}\text{T-Swap} \qquad \frac{m[i] = \text{load } x \quad \mathsf{st}' = (\mathsf{lt}(x) \sqcup \mathsf{pt}) \cdot \mathsf{st}}{m, i \vdash (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \mathsf{st}) \Rightarrow (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \mathsf{st}')}\text{T-Load}$$

$$\frac{m[i] = \text{put } y \quad \mathsf{ht}' = \mathsf{ht}[y \mapsto \tau \sqcup \mathsf{pt}]}{m, i \vdash (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \tau \cdot \mathsf{st}) \Rightarrow (\mathsf{pt}, \mathsf{ht}', \mathsf{lt}, \mathsf{st})}\text{T-Put} \qquad \frac{m[i] = \text{ifeq } j \quad \mathsf{pt}' = \tau \sqcup \mathsf{pt} \quad \mathsf{pt}' = \mathsf{H} \rightarrow i \in \mathtt{PM}_2(m)}{m, i \vdash (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \tau \cdot \mathsf{st}) \Rightarrow (\mathsf{pt}', \mathsf{ht}, \mathsf{lt}, \mathsf{st})}\text{T-If} \qquad \frac{m[i] = \text{goto } j}{m, i \vdash (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \mathsf{st}) \Rightarrow (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \mathsf{st})}\text{T-Goto}$$

$$\frac{m[i] = \text{get } y \quad \mathsf{st}' = (\mathsf{ht}(x) \sqcup \mathsf{pt}) \cdot \mathsf{st}}{m, i \vdash (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \mathsf{st}) \Rightarrow (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \mathsf{st}')}\text{T-Get} \qquad \frac{m[i] = \text{ifneq } j \quad \mathsf{pt}' = \tau \sqcup \mathsf{pt} \quad \mathsf{pt}' = \mathsf{H} \rightarrow i \in \mathtt{PM}_2(m)}{m, i \vdash (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \tau \cdot \mathsf{st}) \Rightarrow (\mathsf{pt}', \mathsf{ht}, \mathsf{lt}, \mathsf{st})}\text{T-Ifn} \qquad \frac{m[i] = \text{return} \quad (\mathsf{ht}, \tau) \models \mathrm{sig}_P(m)}{m, i \vdash (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \tau \cdot \mathsf{st}) \Rightarrow (\mathsf{ht}, \tau)}\text{T-Ret}$$

$$\frac{m[i] = \text{invoke } m' \quad \mathrm{argv}(m') = x_0, \cdots, x_k \quad (\mathsf{pt}_1, \mathsf{ht}_1, \mathsf{lt}_1) \hookrightarrow_{m'} (\mathsf{ht}_2, \tau) \quad \mathsf{pt} \sqsubseteq \mathsf{pt}_1 \quad \mathsf{ht} \sqsubseteq \mathsf{ht}_1 \quad \tau_0 \sqsubseteq \mathsf{lt}_1(x_0) \cdots \tau_k \sqsubseteq \mathsf{lt}_1(x_k) \quad \tau' = \tau \sqcup \mathsf{pt}}{m, i \vdash (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \tau_k \cdots \tau_0 \cdot \mathsf{st}) \Rightarrow (\mathsf{pt}, \mathsf{ht}_2, \mathsf{lt}, \tau' \cdot \mathsf{st})}\text{T-Call}$$

Figure 6: Typing rules

then present an information-flow type system for proving JIT-time-balancing under our protected JIT compilation.

4.1 Protection Mechanism

The first level of our protection mechanism is to disable JIT compilation and inlining of methods which potentially induce leaks. We denote by $\mathtt{PM}_1$ the set of methods that cannot be JIT compiled or inlined, while methods $m' \in \mathbf{M} \setminus \mathtt{PM}_1$ can be JIT compiled or inlined. The second level is to disable JIT optimization of branch points in methods $\mathbf{M} \setminus \mathtt{PM}_1$, whose optimization will potentially induce leaks. We denote by $\mathtt{PM}_2$ the mapping from $\mathbf{M} \setminus \mathtt{PM}_1$ to sets of branch points that cannot be JIT optimized. When the method m is JIT compiled, $\mathtt{PM}_2(m)$ will be updated accordingly.

From the perspective of $\mathrm{JVM}_{\mathrm{JIT}}$ semantics, the compilation directive of any method from $\mathtt{PM}_1$ is limited to $\mathbf{d}_\emptyset$, and the compilation directives of any method $m' \in \mathbf{M} \setminus \mathtt{PM}_1$ can neither inline a method from $\mathtt{PM}_1$ nor optimize the branch at a program point in $\mathtt{PM}_2(m')$.

A *compilation policy* of a program P is given by a pair $(\mathtt{PM}_1, \mathtt{PM}_2)$. A pf-schedule $\mathbf{d}^\star$ that is compliant to the compilation policy $(\mathtt{PM}_1, \mathtt{PM}_2)$ is called a $(\mathtt{PM}_1, \mathtt{PM}_2)$*-schedule*.

4.2 Type System and Inference

We propose an information-flow type system for proving that time-balanced programs are JIT-time-balanced under a fine-grained JIT compilation with a compilation policy $(\mathtt{PM}_1, \mathtt{PM}_2)$.

Lattice for security levels. We consider a lattice of security levels $\mathbb{L} = \{\mathsf{H}, \mathsf{L}\}$ with $\mathsf{L} \sqsubseteq \mathsf{L}$, $\mathsf{L} \sqsubseteq \mathsf{H}$, $\mathsf{H} \sqsubseteq \mathsf{H}$ and $\mathsf{H} \not\sqsubseteq \mathsf{L}$. Initially, all the public inputs have the low security level L and the other inputs have the high security level H. We denote by $\tau_1 \sqcup \tau_2$ the least upper bound of two security levels $\tau_1, \tau_2 \in \mathbb{L}$, namely, $\tau \sqcup \mathsf{H} = \mathsf{H} \sqcup \tau = \mathsf{H}$ for $\tau \in \mathbb{L}$ and $\mathsf{L} \sqcup \mathsf{L} = \mathsf{L}$.

Typing judgments. Our type system supports programs whose control flow depends on secrets. Thus, the typing rules for instructions rely on its path context pt, which indicates whether an instruction is contained in a secret branch. We use functions $\mathsf{ht} : \mathbf{GVar} \rightarrow \mathbb{L}$ and $\mathsf{lt} : \mathbf{LVar} \rightarrow \mathbb{L}$ which map global and local variables to security levels. We also use a stack type (i.e., a stack of security levels) st for typing operand stack. The order $\sqsubseteq$ is lifted to the functions and the stack type as usual, e.g., $\mathsf{ht}_1 \sqsubseteq \mathsf{ht}_2$ if $\mathsf{ht}_1(y) \sqsubseteq \mathsf{ht}_2(y)$ for each $y \in \mathbf{GVar}$.

A *typing judgment* for non-return instructions is of the form $m, i \vdash (\mathsf{pt}_1, \mathsf{ht}_1, \mathsf{lt}_1, \mathsf{st}_1) \Rightarrow (\mathsf{pt}_2, \mathsf{ht}_2, \mathsf{lt}_2, \mathsf{st}_2)$, where m is the method under typing, i is a program point in m. This judgment states that, given the typing context $(\mathsf{pt}_1, \mathsf{ht}_1, \mathsf{lt}_1, \mathsf{st}_1)$, the instruction $m[i]$ yields a new typing context $(\mathsf{pt}_2, \mathsf{ht}_2, \mathsf{lt}_2, \mathsf{st}_2)$. A *typing judgment* of the return is of the form $m, i \vdash (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \mathsf{st}) \Rightarrow (\mathsf{ht}, \tau)$, where ht is the security levels of the global variables and τ is the security level of the return value.

A *security environment* se_m of a method m is a function where for every program point i of m, $\mathrm{se}_m(i)$ is a typing context (ht, τ) if $m[i]$ is a return instruction, and $(\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \mathsf{st})$ otherwise.

Method signature. A (security) *signature* of a method m is of the form $(\mathsf{pt}, \mathsf{ht}_1, \mathsf{lt}_1) \hookrightarrow_m (\mathsf{ht}_2, \tau)$, which states that, given the typing context $(\mathsf{pt}, \mathsf{ht}_1, \mathsf{lt}_1)$, each global variable $y \in \mathbf{GVar}$ has the security level $\mathsf{ht}_2(y)$ and the return value of the method m has the security level τ. Each invocation of m should respect the signature of m. The signature of the program P, denoted by sig_P, is a mapping from the methods of the program P to their signatures. Since a method invoked in any secret branch cannot be JIT compiled or inlined, we require that, for any $m \in \mathbf{M}$, $m \in \mathtt{PM}_1$ if the path context pt in $\mathrm{sig}_P(m)$ is the high security level H.

Typing rules. The typing rules are shown in Figure 6, where the key premises are highlighted and $(\mathsf{ht}, \tau) \models \mathrm{sig}_P(m)$ means that $\mathsf{ht} \sqsubseteq \mathsf{ht}'$ and $\tau \sqsubseteq \tau'$ for $\mathrm{sig}_P(m) = (\mathsf{pt}, \mathsf{ht}_1, \mathsf{lt}_1) \hookrightarrow_m (\mathsf{ht}', \tau')$.

The type system only checks bytecode programs, thus there is no typing rule for the deoptimization instruction deopt md. Rules (T-Push), (T-Pop), (T-Bop) and (T-Swap) track the flow of the secret data via the operand stack. Rules (T-Str), (T-Load), (T-Put) and (T-Get) track the flow of the secret data via local and global variables. Rule (T-Goto) does not change the typing context.

Rules (T-If) and (T-Ifn) require that the path context pt' of each branch has a security level no less than the current path context and the security level of the branching condition on top of the stack. This allows us to track implicit flows during typing. Furthermore, the branch point i should not be optimized by requiring $i \in \mathtt{PM}_2(m)$ if pt' has the high security level H, otherwise the branches may become unbalanced, resulting in JIT-induced leaks.

Rule (T-Ret) requires $(\mathsf{ht}, \tau) \models \mathrm{sig}_P(m)$ that avoids the security levels of the global variables in ht and the security level τ of the return value are greater than these in the method signature $\mathrm{sig}_P(m)$.

Rule (T-Call) ensures that the context of invoke m' meets the signature $\mathrm{sig}_P(m') = (\mathsf{pt}_1, \mathsf{ht}_1, \mathsf{lt}_1) \hookrightarrow_{m'} (\mathsf{ht}_2, \tau)$, e.g., $\mathsf{pt} \sqsubseteq \mathsf{pt}_1$ avoiding that the current path context pt has a security level greater than the excepted one pt_1, and $\tau_0 \sqsubseteq \mathsf{lt}_1(x_0) \cdots \tau_k \sqsubseteq \mathsf{lt}_1(x_k)$ avoiding that actual arguments have the security levels greater than that of formal arguments.

Typable methods. The JIT-time-balancing is verified by type inference. To formalize this, we first introduce some notations [14].

Let us fix a method m. For each program point i, let $\mathsf{nxt}_m(i)$ be the set of successors of i w.r.t. the control flow. Formally, $\mathsf{nxt}_m(i) = \{j\}$ if $m[i]$ is goto j, $\mathsf{nxt}_m(i) = \{i+1, j\}$ if $m[i]$ is ifeq j or ifneq j, $\mathsf{nxt}_m(i) = \emptyset$ if $m[i]$ is return, and $\mathsf{nxt}_m(i) = \{i+1\}$ otherwise.

For each branch point i, let $\mathsf{junc}(i)$ denote its junction point, i.e., the immediate post-dominator of i. (Recall that we assumed there is no early return in branches, thus $\mathsf{junc}(i)$ is well-defined.) We denote by $\mathsf{region}(i)$ the set of program points j that can be reached from the branch point i and are post-dominated by $\mathsf{junc}(i)$. We denote by $\mathsf{maxBP}(j)$ the set of branch points i such that $j = \mathsf{junc}(i)$ and $\mathsf{region}(i) \not\subset \mathsf{region}(i')$ for any $i' \in \mathsf{maxBP}(j)$. Intuitively, $\mathsf{maxBP}(j)$ contains the branch points i with the junction point j and $\mathsf{region}(i)$ is not contained by $\mathsf{region}(i')$ of any other branch point i' with the same junction point j, namely, nested branch points i' of the branch point i are excluded.

The method m is *typable* w.r.t. sig_P and $(\mathtt{PM}_1, \mathtt{PM}_2)$, denoted by $(\mathtt{PM}_1, \mathtt{PM}_2, \mathsf{sig}_P) \rhd m$, if there exists a security environment se_m such that $\mathsf{se}_m(0) = (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \epsilon)$ for $\mathsf{sig}_P(m) = (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}) \hookrightarrow_m (\mathsf{ht}', \tau)$ and one of the following conditions holds for each program point i:

- if i is not a junction point, then $m, j \vdash \mathsf{se}_m(j) \Rightarrow \mathsf{se}_m(i)$ for the program point j such that $\mathsf{nxt}_m(j) = \{i\}$;
- if i is a junction point, suppose $\mathsf{se}_m(i) = (\mathsf{pt}, \mathsf{ht}, \mathsf{lt}, \mathsf{st})$, then the following two conditions hold:
  - there exists some $j \in \mathsf{maxBP}(i)$ with $\mathsf{pt}' \sqsubseteq \mathsf{pt}$ and $\mathsf{se}_m(j) = (\mathsf{pt}', \mathsf{ht}', \mathsf{lt}', \mathsf{st}')$;
  - $\mathsf{ht} \sqsubseteq \mathsf{ht}'$, $\mathsf{lt} \sqsubseteq \mathsf{lt}'$ and $\mathsf{st} \sqsubseteq \mathsf{st}'$ for $\mathsf{nxt}(j) = i$ and $\mathsf{se}_m(j) = (\mathsf{pt}', \mathsf{ht}', \mathsf{lt}', \mathsf{st}')$.

Intuitively, $(\mathtt{PM}_1, \mathtt{PM}_2, \mathsf{sig}_P) \rhd m$ requires that (i) secret branches are forbidden to be optimized by $\mathtt{PM}_2$ and (ii) methods m' invoked in $\mathsf{region}(i)$ of any secret branches $m[i]$ are forbidden to be JIT compiled and inlined. Recall that we have assumed $m' \in \mathtt{PM}_1$ if the path context pt in $\mathsf{sig}_P(m')$ has the high security level H.

A program P is *typable* w.r.t. sig_P and $(\mathtt{PM}_1, \mathtt{PM}_2)$, denoted by $(\mathtt{PM}_1, \mathtt{PM}_2, \mathsf{sig}_P) \rhd P$, if (i) the signature sig_m of the entry point m is $(\mathsf{L}, \mathsf{ht}, \mathsf{lt}) \hookrightarrow_m (\mathsf{ht}', \tau)$ such that $\mathsf{ht}(y) = \mathsf{H}$ and $\mathsf{lt}(x) = \mathsf{H}$ for any secret inputs x, y; and (ii) $(\mathtt{PM}_1, \mathtt{PM}_2, \mathsf{sig}_P) \rhd m$ for every $m \in \mathbf{M}$.

Theorem 4.1. *If program P is time-balanced and $(\mathtt{PM}_1, \mathtt{PM}_2, sig_P) \rhd P$, then P is JIT-time-balanced under $(\mathtt{PM}_1, \mathtt{PM}_2)$-schedules.*

The proof is provided in the technical report [56]. Note that the native code in the code heap of each initial configuration can only be complied from bytecode following the policy $(\mathtt{PM}_1, \mathtt{PM}_2)$.

5 IMPLEMENTATION FOR PRACTICAL JAVA

We have implemented our approach as a tool DeJITLeak for real-world Java bytecode (Jar packages). DeJITLeak consists of two main components: type inference for computing a signature sig_P and a policy $(\mathtt{PM}_1, \mathtt{PM}_2)$ such that $(\mathtt{PM}_1, \mathtt{PM}_2, \mathsf{sig}_P) \rhd P$, and a protection mechanism in HotSpot from OpenJDK [53].

5.1 Type Inference

Our type inference is built on JOANA [40], a sound, flow-, context-, and object-sensitive information flow framework based on program dependence graphs (PDGs). Given a program P annotated with public inputs, we first identify secret inputs and then leverage JOANA to compute a security environment se_m and a signature $\mathsf{sig}_P(m)$ for each method m via solving flow equations. We then locate all the branch points in each method m whose path context or branching condition has the high security level H, namely, all the secret branches. These branch points are added in $\mathtt{PM}_2(m)$, as they can potentially induce Topti and Tbran leaks when JIT optimized.

From the branch points $\mathtt{PM}_2(m)$, we identify and extract all the methods invoked within $\mathsf{region}(i)$ for all the branch points $i \in \mathtt{PM}_2(m)$. These methods can potentially induce Tmeth leaks when JIT compiled or inlined, thus, are added in $\mathtt{PM}_1$. According to our type system and the soundness of JOANA, the program P is typable w.r.t. sig_P and $(\mathtt{PM}_1, \mathtt{PM}_2)$, i.e., $(\mathtt{PM}_1, \mathtt{PM}_2, \mathsf{sig}_P) \rhd P$ holds.

5.2 Protection Mechanism in HotSpot

To enforce a compilation policy $(\mathtt{PM}_1, \mathtt{PM}_2)$ during JIT compilation, we modify HotSpot to demonstrate our approach. To prevent a method $m \in \mathtt{PM}_1$ from being compiled and inlined, we use the option `CompileCommand` supported by HotSpot [52], namely,

```
-XX:CompileCommand=exclude, signature_of_the_method
-XX:CompileCommand=dontinline, signature_of_the_method
```

where the option `exclude` disables JIT compilation of the method `signature_of_the_method`, and `dontinline` prevents the method `signature_of_the_method` from procedure inline.

Unfortunately, HotSpot does not provide any option that can be used to specify branch points where branch prediction and/or optimistic compilation can be disabled. Therefore, we modified HotSpot to support an additional command `dontprune` that allows us to specify branch points. The command `dontprune` is used similar to `exclude`, but with an additional list of branch points for the specified method. During JIT compilation, both branch prediction and optimistic compilation are prohibited for all these branch points, even if the method is recompiled.

5.3 DeJITLeak$_{\texttt{light}}$

To reduce performance overhead, we also propose and implement an alternative protection mechanism DeJITLeak$_{\texttt{light}}$. It only disables the inlining of the methods $m \in \mathtt{PM}_1$ whereas DeJITLeak disables both JIT compilation and inlining of the methods $m \in \mathtt{PM}_1$. This weaker protection mechanism is still sound under the assumption that the methods invoked on both sides of each secret branch point are the same. This assumption is reasonable in practice, as it is a straightforward for developers to implement a time-balanced program by invoking same methods in both sides of each secret branch point. Remark that inlining method should be disabled even if a method is invoked on both sides of a secret branch point, as the method may be inlined only in one branch, inducing leaks.

6 EVALUATION

We first evaluate the efficiency of the type inference and then compare our approach with other strategies: NOJIT, DisableC2, and MExclude (cf. Section 2.2). According to [16], we only disable JIT compilation of the methods that contain some secret branch points for MExclude. Finally, we conduct a case study.

Experiment setup. We evaluate DeJITLeak and DeJITLeak$_{\texttt{light}}$ on the benchmarks used in recent side-channel detection: Blazer [6],

Themis [23], CoCo-Channel [18], DifFuzz [51], and JVMFuzz [19], including real-world programs from well-known Java applications such as Apache FtpServer, micro-benchmarks from DARPA STAC and classic examples from the literature [36, 43, 54]. Recall that we target time-balanced Java bytecode. Thus, we only consider the "safe" versions, i.e., programs that are leakage-free or only have slight leaks under their leakage models without JIT compilation. We also exclude the benchmarks *tomcat*, *pac4j*, and *tourplanner* from Themis, as *tomcat* and *pac4j* have significant leakages [19] while *tourplanner* is time-consuming (0.5 hour per execution and we shall run each benchmark 1,000 times per branch). The remaining benchmarks are shown in Table 1, where ♯LOC shows the number of lines in the Java source code counted by cloc [24]. Note that for the purpose of experiments, *k96**, *modpow1** and *modpow2** are patched versions of *k96*, *modpow1* and *modpow2*, and *unixlogin* is a patched version by DifFuzz to resolve the NullPointerException error in its original version from Blazer.

All experiments are conducted on an Intel NUC running Ubuntu 18.04 with Intel Core I5-8259U CPU @ 2.30GHz and 16GB of memory. To be practical, we do not disable CPU-level and other JIT optimizations when JIT compilation is enabled.

In summary, the results show that (1) DeJITLeak is more effective than DisableC2 and MExclude on almost all the benchmarks, and (2) DeJITLeak$_{\texttt{light}}$ is able to achieve comparable effectiveness as DeJITLeak and induces significantly less performance loss.

6.1 Results of Type Inference

Table 1 shows the results, where columns ♯Node and ♯Edge show the number of nodes and edges in the corresponding PDG on which type inference is performed, column T_{taint} (ms) shows the execution time of type inference, column T_{total} (s) gives the overall execution time and column Mem (Mb) gives the overall memory consumption. We observe that these benchmarks can be solved efficiently. It takes 1.99 seconds on average (up to 5.52 seconds) and 254 Mb for one benchmark. Note that the overall time and memory consumption does not necessarily correlate with ♯LOC (e.g., on *gpt14* vs. *k96*), because we only counted the number of lines in the Java source code but excluded the code of invoked methods from libraries which were also analyzed during type inference. We note that the time and memory of analyzing a hello world program without taint source is 0.93 seconds and 161 Mb.

6.2 Effectiveness and Efficiency

We evaluate the effectiveness by quantifying the amount of leakages in practice using mutual information [47], a widely used metric for side channel analysis [44, 45, 48, 59]. The mutual information of a program containing a vulnerable conditional statement with the secret condition K and execution time T is defined as $\mathrm{I}(K;T) = \mathrm{H}(K) - \mathrm{H}(K|T)$, where $\mathrm{H}(K)$ is classical Shannon entropy measuring uncertainty about K, and $\mathrm{H}(K|T)$ is the conditional Shannon entropy of K given T. $\mathrm{I}(K;T)$ measures the uncertainty about K after an attacker has learned the execution time T. We create attacks to explore the maximum amount of leakages according to [17]. To discretize the execution time T, we split it into a 20 bins. Note that the closer the mutual information value is to 1, the stronger the relationship between the branch condition K and execution time T.

Table 1: Results of type inference

	Name	♯LOC	♯Node	♯Edge	T_{taint} (ms)	T_{total} (s)	Mem (Mb)
DifFuzz	clear	13	11	20	20.24	1.63	242
	md5	13	46	67	24.47	2.58	249
	salted	13	51	82	26.13	2.59	259
	stringutils	194	15	25	21.52	1.66	244
	authmreloaded	19	45	62	23.91	4.02	396
Blazer	array	35	2	1	16.21	1.00	164
	gpt14	51	17	22	38.15	2.01	317
	k96	40	21	33	51.43	2.16	331
	login	53	2	1	17.33	0.99	164
	loopbranch	48	2	1	16.55	0.97	160
	modpow1	141	23	35	49.54	2.13	294
	modpow2	106	14	23	31.80	2.00	312
	passwordEq	38	5	6	18.81	1.55	237
	sanity	30	2	1	16.36	0.96	160
	straightline	32	2	1	16.72	1.01	163
	unixlogin	45	21	29	22.65	1.26	193
Themis	bootauth	125	21	34	23.89	3.80	340
	jdk	23	2	1	16.66	0.99	164
	jetty	32	4	5	18.40	1.54	233
	orientdb	211	61	97	39.01	5.52	453
	picketbox	47	6	7	18.57	1.58	238
	spring	39	7	7	19.90	1.75	277

The results are reported in Table 2 in the average of 1,000 experiments for each benchmark, where the best results among different methods are in **bold face**. The second and third columns show the leakage and execution time without any defense. The other columns show the leakage with the corresponding defense and the overhead (calculated as the ratio: execution time with the defense/execution time without defense) induced by the defense.

Effectiveness. Overall, we can observe that (1) all these "safe" programs are vulnerable (i.e., nonnegligible leakage) due to JIT compilation; (2) disabling JIT compilation (NOJIT) can effectively reduce JIT-induced leakages for most programs except for *array*, *login*, *loopbranch*, *straightline* and *unixlogin*; (3) DeJITLeak and DeJITLeak$_{\texttt{light}}$ perform significantly better than DisableC2 and MExculde, even better than NOJIT on some benchmarks (e.g., *md5*, *array*, *login*, *loopbranch*, *sanityjdk* and *jetty*); (4) DeJITLeak and DeJITLeak$_{\texttt{light}}$ are almost comparable.

Efficiency. We measure the efficiency of respective approach by the times the execution time is increased. In general, (1) NOJIT incurs the highest performance cost; (2) DisableC2 and MExclude lead to nearly 2–7 times runtime overhead; (3) DeJITLeak incurs more overhead than DisableC2 and MExclude, but still outperforms DisableC2 and MExclude on many benchmarks; (4) DeJITLeak$_{\texttt{light}}$ brings the least runtime overhead (up to 1.82 times).

On some benchmarks (e.g., *authmreloaded*, *array*, *login*, *loopbranch*, *sanity* and *jdk*), DeJITLeak performs better than DisableC2. It is because DisableC2 completely disables C2 mode compilation for all the methods, whereas DeJITLeak disables JIT compilation and procedure inline of methods invoked in secret branches. Thus, DeJITLeak performs better than DisableC2 when many methods can be compiled in the C2 mode at runtime. We note that MExclude allows JIT compilation and inlining of methods invoked in secret branches, thus outperforms DeJITLeak in general. When many methods contain secret branches but few methods are invoked therein, DeJITLeak performs better than MExclude.

Table 2: Evaluation results of DeJITLeak and DeJITLeak$_{\texttt{light}}$

	Benchmark			NOJIT		DisableC2		MExclude		DeJITLeak		DeJITLeak$_{\texttt{light}}$	
	Name	Leakage	Time (µs)	Leakage	Overhead	Leakage	Overhead	Leakage	Overhead	Leakage	Overhead	Leakage	Overhead
DifFuzz	clear	1.00	4.846	0.02	49.40	0.02	3.47	0.02	12.95	**0.01**	25.22	1.00	**1.00**
	md5	1.00	6.526	0.19	47.81	0.09	4.13	**0.01**	10.00	**0.01**	19.51	**0.01**	**1.82**
	salted	1.00	6.711	**0.02**	47.80	0.17	3.93	0.20	9.69	0.03	18.99	0.17	**1.77**
	stringutils	0.97	0.559	**0.10**	11.90	0.59	1.57	1.00	2.64	0.77	8.92	1.00	**1.35**
	authmreloaded	1.00	8.696	**0.01**	34.89	0.05	4.46	0.03	1.28	0.03	**1.00**	0.03	**1.00**
	Average	0.99	5.468	**0.07**	38.36	0.18	3.51	0.25	7.31	0.17	14.73	0.44	**1.39**
Blazer	array	1.00	0.229	1.00	2.00	0.64	1.21	1.00	2.61	**0.23**	**1.00**	0.25	**1.00**
	gpt14	1.00	2.157	**0.01**	45.11	**0.01**	3.06	0.20	1.80	**0.01**	15.95	**0.01**	**1.47**
	k96	1.00	2.414	**0.02**	42.69	1.00	3.04	0.79	1.83	1.00	18.50	1.00	**1.46**
	k96*	1.00	2.372	**0.02**	42.93	**0.02**	3.09	0.59	1.90	**0.02**	18.99	0.52	**1.48**
	login	1.00	0.266	0.79	2.05	0.67	1.17	0.91	2.68	**0.54**	**1.05**	**0.54**	**1.05**
	loopbranch	1.00	0.243	0.86	5.57	0.80	3.15	0.33	15.34	**0.01**	**0.98**	**0.01**	**0.98**
	modpow1	1.00	78.615	**0.02**	0.36	1.00	0.21	1.00	0.65	1.00	**0.16**	1.00	0.95
	modpow1*	1.00	78.542	**0.01**	0.36	0.02	0.23	1.00	0.65	**0.01**	**0.16**	**0.01**	0.94
	modpow2	1.00	0.789	**0.01**	36.92	1.00	2.78	1.00	2.27	1.00	15.61	1.00	**1.57**
	modpow2*	1.00	0.945	0.01	42.15	0.07	2.93	1.00	2.12	0.01	17.55	**0.00**	**1.55**
	passwordEq	1.00	0.262	0.13	6.61	0.17	1.53	0.56	3.74	**0.01**	5.39	**0.01**	**1.15**
	sanity	1.00	0.234	0.25	5.83	0.97	2.82	0.07	16.02	**0.01**	**0.99**	**0.01**	1.00
	straightline	1.00	0.231	0.80	2.03	0.07	1.07	0.90	2.16	**0.00**	**1.00**	0.01	**1.00**
	unixlogin	1.00	0.316	**1.00**	8.51	**1.00**	1.96	**1.00**	3.03	**1.00**	10.09	**1.00**	**1.37**
	Average	1.00	11.973	**0.35**	17.37	0.53	2.02	0.74	4.06	**0.35**	7.67	0.38	**1.21**
Themis	bootauth	1.00	2.793	0.02	106.98	**0.01**	4.53	0.03	1.53	0.84	1.47	0.04	**1.05**
	jdk	1.00	0.236	0.16	2.15	0.05	1.14	0.19	2.68	**0.01**	**1.01**	**0.01**	**1.01**
	jetty	1.00	0.254	0.11	6.49	0.17	1.51	0.50	3.51	**0.01**	5.48	**0.01**	**1.14**
	orientdb	0.99	1.942	**0.01**	78.48	0.01	3.47	0.33	1.39	**0.01**	1.28	**0.01**	**0.99**
	picketbox	1.00	0.252	0.04	7.23	0.02	1.54	1.00	1.82	0.06	7.85	**0.01**	**1.30**
	spring	1.00	0.509	**0.01**	14.16	0.02	2.10	0.04	2.63	**0.01**	1.71	**0.01**	**1.06**
	Average	1.00	0.998	0.06	35.92	0.05	2.38	0.35	2.26	0.16	3.13	**0.02**	**1.09**

6.3 Case Study

We discuss some interesting case studies below.

array, login, loopbranch, straightline, unixlogin: Results show that their leakages are significant in practice, although they are "safe" benchmarks without JIT compilation [6, 23, 51]. We found that *array*, *login*, *loopbranch*, and *straightline* have balanced branches in terms of the number of instructions at the source code level. However, a branch with a balanced number of instructions does not necessarily have balanced execution time even if JIT is disabled. This indicates that modeling time-balancing using the number of instructions may not be precise. Interestingly, both DeJITLeak and DeJITLeak$_{\texttt{light}}$ are able to significantly reduce the leakage of *array*, *login*, *loopbranch*, and *straightline*. This is because the percentage of timing difference is fixed, the program speeds up with the JIT compilation (i.e., lower overhead), making side channel-unstable and difficult to observe due to the fixed noise. The case for *unixlogin* is slightly different. Recall that *unixlogin* is a patched version by DifFuzz to resolve the NullPointerException error in its original version from Blazer. However, this patch introduced a leakage which is always significantly observable.

stringutils: We observe that only NOJIT effectively reduces the JIT-induced leakage of *stringutils*. We found that *stringutils* evaluates a method in Apache FtpServer that pads a string to a specified length, where an insecure version would leak information about the original string's length. DeJITLeak and DeJITLeak$_{\texttt{light}}$ successfully eliminated the JIT-induced leak in this method, guaranteeing the balance of secret branches in the native code. However, due to CPU-level optimizations (e.g., speculative execution), the execution time of different branches varies with secret inputs.

k96, modpow1, modpow2: Similar to *stringutils*, we observe that only NOJIT effectively reduces their JIT-induced leakages. These programs implement various components of the RSA cryptosystem's modular exponentiation using the classic square-and-multiply algorithm, thus their leakages would result in key recovery attacks [43]. DeJITLeak and DeJITLeak$_{\texttt{light}}$ indeed can guarantee that no leaks are induced by JIT compilation in the native code. However, due to CPU-level optimizations, the execution time of the branches varies with secret inputs. To reduce such noise, we created patched versions *k96**, *modpow1** and *modpow2** by moving the time-consuming operations from branches to outside of their branching point. After patching, most defense solutions are able to reduce the JIT induced leakages.

bootauth: DeJITLeak is not effective on *bootauth*, due to an unbalanced branching statement in bytecode. According to our policy, we need to disable JIT compilation for methods (i.e., *fromJSON*, *getTime* and *getExpires*) invoked in secret-dependent branches, but other methods can be JIT complied including the C2 mode compilation. But this unfortunately amplifies the timing difference of the existing leak, compared over the entire execution time. Remark that our approach is designed for programs that are time-balanced at the bytecode level.

6.4 Discussion

Limitations. First, the execution of JVM profiling, JIT compilation and garbage collection that may affect the overall execution

time. We did not formalize them, as they are often executed asynchronously in different threads and are difficult to be exploited. To our knowledge, no attack leverages them. Second, we did not formalism CPU-level (such as speculative execution and cache) and other JIT optimizations (e.g., constant propagation, loop unfolding and dead elimination) that may induce timing side-channel leaks. Such leaks have been considered for statically compiled binary code [25]. To our knowledge, no existing attack leverages those optimizations for Java programs. JIT-induced leaks are significant to be exploited remotely in real-world applications, thus, for the sake of separating concerns, we do not consider other leakages. To detect and mitigate timing side-channel leaks induced by CPU-level optimizations, one may combine our approach with existing ones, e.g., [25, 26, 41, 61, 63, 68, 69]. This is a future work to be explored.

Threats. The first main threat to our evaluation is the noise of execution time introduced by the compiler (e.g., JVM profiling, JIT compilation, and garbage collection) and hardware (e.g., CPU-level optimizations). To mitigate the threat, we run each benchmark 1,000 times per branch in real-world JVM HotSpot without disabling CPU-level and other JIT optimizations. The second main threat to our evaluation is the small benchmarks and non-interference from other users. In this setting, the attacks are more powerful, namely, the adversary is able to measure and prime accurately, and thus are more difficult to defeat. We do not impose a bound on the attacker's ability, therefore provide theoretic security guarantees. In practice, the timing measurement would be less undistinguishable and the prime would be more difficult due to significant interference from other users or the JIT itself on large programs. Therefore, the evaluation results in such a setting should be validated in future.

7 RELATED WORK

Timing side-channel attacks have attracted many attentions, with a significant amount of work devoted to its detection [18, 48, 51, 54], verification [4, 6, 10, 15, 23, 25, 27, 28] and mitigation [1, 22, 27, 49, 65, 66], which vary in targeted programs, leakage models, techniques, efficiency and precision, etc.

More recent work focuses on other sources of timing side-channels, induced by micro-architectural features (e.g., Spectre [42] and Meltdown [46]) or compilation (e.g., JIT-induced leaks [17]) where provably leakage-free programs (or with slight leakages) may become vulnerable when they are taken into account. Our work is within this category.

Micro-architectural features allow new timing side-channel attacks such as Spectre, Meltdown and variants thereof [20, 21, 50, 57, 60]. This problem has been recently studied [21, 26, 37–39, 41, 61, 63, 67–69], where speculative execution semantics, notions of constant-time under the new semantics, detection and mitigation approaches, etc, have been proposed. Among them, Blade [61] is the closest to our work, which aims to ensure that constant-time programs are leakage-free under speculative and out-of-order execution. Our work is similar in spirit, but as the leaks induced by JIT compilation and micro-architecture features are different, the concrete technology (e.g., security notions, detection and mitigation approaches) in this paper is new. Moreover, as discussed in our experiments, native code compiled from bytecode may suffer from leakages induced by micro-architectural features. Such leakages could potentially be eliminated by integrating existing mitigation approaches (e.g., Blade) into JIT compilation.

Besides JIT compilation, static compilation can also introduce timing leakages. To address this problem, constant-time preserving compilation has been studied [13] and subsequently implemented in the verified compiler CompCert [11]. However, they disallow secret branches, increasing the difficulty of implementing constant-time programs. Follow-up work includes constant-resource preserving compilation [12] and timing side-channel security analysis of binary code [25]. However, none of them considered JIT compilation which is far more complex than the static compilation.

The work on JIT-induced timing channel is currently very limited. The work close to ours is [16, 17, 19]. The JIT-induced leaks proposed in [17] demonstrated how JIT compilation can be leveraged to mount timing side-channel attacks. A fuzzing approach was proposed to detect JIT-induced leaks [19]. However, it may report false negatives and cannot mitigate JIT-induced leaks. The three strategies (i.e., NOJIT, DisableC2 and MExclude) proposed in [16] have been discussed and compared in Section 2.2 and Section 6.2.

In addition to detecting and mitigating timing side-channel attacks, there are techniques for detecting and mitigating power side-channel attacks [9, 29, 32–35, 64, 70] and attacks against secure multi-party computation [5, 30] where the adversary is able to observe all the public information during computation. Each type of attack has unique characteristics, in general, these existing techniques are orthogonal to our work.

8 CONCLUSION

In this paper, we formalized time-balancing under JIT compilation, based on which we proposed an automated approach to eliminate JIT-induced leaks. Our approach systematically detects potential leaks via a precise information flow analysis and eliminates potential leaks via a fine-grained JIT compilation. We implemented our approach in the tool DeJITLeak for real-world Java programs. The evaluation showed that DeJITLeak is more effective than existing solutions and provides a trade-off between security and performance. The lightweight variant DeJITLeak$_{\texttt{light}}$ of DeJITLeak further reduces the overhead but with comparable effectiveness.

In future, we plan to improve our approach by taking into account other JIT optimizations and CPU-level optimizations that also introduce timing side-channels in practice.

DATA-AVAILABILITY STATEMENT

To foster further research, benchmarks, experimental data and the prototyping tool are released at [55].

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (NSFC) under Grants No. 62072309 and No. 61872340, an oversea grant from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2018A16), and Birkbeck BEI School Project (EFFECT).

REFERENCES

[1] Johan Agat. 2000. Transforming Out Timing Leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 40–53. https://doi.org/10.1145/325694.325702

[2] Martin R. Albrecht and Kenneth G. Paterson. 2016. Lucky Microseconds: A Timing Attack on Amazon's s2n Implementation of TLS. In *Proceedings of the 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. 622–643. https://doi.org/10.1007/978-3-662-49890-3_24

[3] Nadhem J. AlFardan and Kenneth G. Paterson. 2013. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (S&P)*. 526–540. https://doi.org/10.1109/SP.2013.42

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*. 53–70.

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Hugo Pacheco, Vitor Pereira, and Bernardo Portela. 2018. Enforcing Ideal-World Leakage Bounds in Real-World Secret Sharing MPC Frameworks. In *Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF)*. 132–146.

[6] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. Decomposition Instead of Self-Composition for Proving the Absence of Timing Channels. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 362–375. https://doi.org/10.1145/3062341.3062378

[7] Konstantinos Athanasiou, Byron Cook, Michael Emmi, Colm MacCárthaigh, Daniel Schwartz-Narbonne, and Serdar Tasiran. 2018. SideTrail: Verifying Time-Balancing of Cryptosystems. In *Proceedings of the 10th International Conference on Verified Software. Theories, Tools, and Experiments* -. 215–228. https://doi.org/10.1007/978-3-030-03592-1_12

[8] Aurèle Barrière, Sandrine Blazy, Olivier Flückiger, David Pichardie, and Jan Vitek. 2021. Formally Verified Speculation and Deoptimization in a JIT Compiler. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–26. https://doi.org/10.1145/3434327

[9] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. 2016. Strong Non-Interference and Type-Directed Higher-Order Masking. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 116–129.

[10] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. 2014. System-Level Non-Interference for Constant-Time Cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1267–1279. https://doi.org/10.1145/2660267.2660283

[11] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2020. Formal Verification of a Constant-Time Preserving C Compiler. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 7:1–7:30. https://doi.org/10.1145/3371075

[12] Gilles Barthe, Sandrine Blazy, Rémi Hutin, and David Pichardie. 2021. Secure Compilation of Constant-Resource Programs. In *Proceedings of the 34th IEEE Computer Security Foundations Symposium (CSF)*. 1–12. https://doi.org/10.1109/CSF51468.2021.00020

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

[14] Gilles Barthe, David Pichardie, and Tamara Rezk. 2013. A Certified Lightweight Non-Interference Java Bytecode Verifier. *Mathematical Structures in Computer Science* 23, 5 (2013), 1032–1081. https://doi.org/10.1017/S0960129512000850

[15] Sandrine Blazy, David Pichardie, and Alix Trieu. 2019. Verifying Constant-Time Implementations by Abstract Interpretation. *Journal of Computer Security* 27, 1 (2019), 137–163. https://doi.org/10.3233/JCS-181136

[16] Tegan Brennan. 2020. *Static and Dynamic Side Channels in Software.* Ph. D. Dissertation. UC Santa Barbara.

[17] Tegan Brennan, Nicolás Rosner, and Tevfik Bultan. 2020. JIT Leaks: Inducing Timing Side Channels through Just-In-Time Compilation. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (S&P)*. 1207–1222. https://doi.org/10.1109/SP40000.2020.00007

[18] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2018. Symbolic Path Cost Analysis for Side-channel Detection. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE)*. 424–425. https://doi.org/10.1145/3183440.3195039

[19] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2020. JVM Fuzzing for JIT-induced Side-Channel Detection. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. 1011–1023. https://doi.org/10.1145/3377811.3380432

[20] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*. 991–1008.

[21] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-Time Foundations for the New Spectre Era. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 913–926. https://doi.org/10.1145/3385412.3385970

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

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

[24] Al Danial. 2021. Count Lines of Code. https://github.com/AlDanial/cloc.

[25] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2020. Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*. 1021–1038. https://doi.org/10.1109/SP40000.2020.00074

[26] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2021. Hunting the Haunter - Efficient Relational Symbolic Execution for Spectre with Haunted RelSE. In *Proceedings of the 28th Annual Network and Distributed System Security Symposium*.

[27] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *Proceedings of the 22th USENIX Security Symposium (USENIX Security)*. 431–446.

[28] Goran Doychev and Boris Köpf. 2017. Rigorous Analysis of Software Countermeasures against Cache Attacks. In *Proceedings of the 38th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 406–421. https://doi.org/10.1145/3062341.3062388

[29] Hassan Eldib, Chao Wang, and Patrick Schaumont. 2014. Formal Verification of Software Countermeasures against Side-Channel Attacks. *ACM Trans. Softw. Eng. Methodol.* 24, 2 (2014), 11:1–11:24.

[30] Yuxin Fan, Fu Song, Taolue Chen, Liangfeng Zhang, and Wanwei Liu. 2022. PoS4MPC: Automated Security Policy Synthesis for Secure Multi-Party Computation. In *Proceedings of the 34th International Conference on Computer Aided Verification (CAV)*.

[31] Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. 2018. Correctness of Speculative Optimizations with Dynamic Deoptimization. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 49:1–49:28. https://doi.org/10.1145/3158137

[32] Pengfei Gao, Hongyi Xie, Fu Song, and Taolue Chen. 2021. A Hybrid Approach to Formal Verification of Higher-Order Masked Arithmetic Programs. *ACM Trans. Softw. Eng. Methodol.* 30, 3 (2021), 26:1–26:42.

[33] Pengfei Gao, Hongyi Xie, Pu Sun, Jun Zhang, Fu Song, and Taolue Chen. 2022. Formal Verification of Masking Countermeasures for Arithmetic Programs. *IEEE Trans. Software Eng.* 48, 3 (2022), 973–1000.

[34] Pengfei Gao, Hongyi Xie, Jun Zhang, Fu Song, and Taolue Chen. 2019. Quantitative Verification of Masked Arithmetic Programs Against Side-Channel Attacks. In *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Held as Part of the European Joint Conferences on Theory and Practice of Software (TACAS)*. 155–173.

[35] Pengfei Gao, Jun Zhang, Fu Song, and Chao Wang. 2019. Verifying and Quantifying Side-channel Resistance of Masked Software Implementations. *ACM Trans. Softw. Eng. Methodol.* 28, 3 (2019), 16:1–16:32.

[36] Daniel Genkin, Itamar Pipman, and Eran Tromer. 2014. Get Your Hands Off My Laptop: Physical Side-Channel Key-Extraction Attacks on PCs. In *Proceedings of the 16th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Vol. 8731. 242–260. https://doi.org/10.1007/978-3-662-44709-3_14

[37] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (S&P)*. 1–19. https://doi.org/10.1109/SP40000.2020.00011

[38] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. 2020. SpecuSym: Speculative Symbolic Execution for Cache Timing Leak Detection. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. 1235–1247. https://doi.org/10.1145/3377811.3380428

[39] Shengjian Guo, Yueqi Chen, Jiyong Yu, Meng Wu, Zhiqiang Zuo, Peng Li, Yueqiang Cheng, and Huibo Wang. 2020. Exposing Cache Timing Side-Channel Leaks Through Out-Of-Order Symbolic Execution. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 147:1–147:32. https://doi.org/10.1145/3428215

[40] Christian Hammer and Gregor Snelting. 2009. Flow-Sensitive, Context-Sensitive, and Object-Sensitive Information Flow Control Based on Program Dependence Graphs. *International Journal of Information Security* 8, 6 (2009), 399–422. https://doi.org/10.1007/s10207-009-0086-1

[41] Zecheng He, Guangyuan Hu, and Ruby B. Lee. 2021. New Models for Understanding and Reasoning about Speculative Execution Attacks. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 40–53. https://doi.org/10.1109/HPCA51647.2021.00014

[42] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz,

and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (S&P)*. 1–19. https://doi.org/10.1109/SP.2019.00002

[43] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*. 104–113. https://doi.org/10.1007/3-540-68697-5_9

[44] Boris Köpf and David A. Basin. 2007. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security (CCS)*. 286–296. https://doi.org/10.1145/1315245.1315282

[45] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2012. Automatic Quantification of Cache Side-Channels. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*, Vol. 7358. 564–580. https://doi.org/10.1007/978-3-642-31424-7_40

[46] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*. 973–990.

[47] Pasquale Malacaria and Jonathan Heusser. 2010. Information Theory and Security: Quantitative Information Flow. In *The 10th International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM)*. 87–134. https://doi.org/10.1007/978-3-642-13678-8_3

[48] Pasquale Malacaria, M. H. R. Khouzani, Corina S. Pasareanu, Quoc-Sang Phan, and Kasper Søe Luckow. 2018. Symbolic Side-Channel Analysis for Probabilistic Programs. In *Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF)*. 313–327. https://doi.org/10.1109/CSF.2018.00030

[49] Heiko Mantel and Artem Starostin. 2015. Transforming Out Timing Leaks, More or Less. In *Proceedings of the 20th European Symposium on Research in Computer Security (ESORICS)*. 447–467. https://doi.org/10.1007/978-3-319-24174-6_23

[50] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Frank Piessens, Berk Sunar, and Yuval Yarom. 2019. Fallout: Reading Kernel Writes From User Space. *CoRR* abs/1905.12701 (2019). http://arxiv.org/abs/1905.12701

[51] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. 2019. DiffFuzz: Differential Fuzzing for Side-Channel Analysis. In *Proceedings of the ACM/IEEE 41st International Conference on Software Engineering (ICSE)*. 176–187. https://doi.org/10.1109/ICSE.2019.00034

[52] Oracle. 2021. HotSpot VM. https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html.

[53] Oracle. 2021. OpenJDK: JDK 8 source code (Mercurial repository), tag jdk8u292-ga. https://hg.openjdk.java.net/jdk8u/jdk8u/jdk.

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

[55] Qi Qin, JulianAndres JiYang, Fu Song, Taolue Chen, and Xinyu Xing. 2022. *DeJITLeak: v1.1.* https://doi.org/10.5281/zenodo.7080369

[56] Qi Qin, JulianAndres JiYang, Fu Song, Taolue Chen, and Xinyu Xing. 2022. Preventing Timing Side-Channels via Security-Aware Just-In-Time Compilation. *CoRR* abs/2202.13134 (2022).

[57] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 753–768. https://doi.org/10.1145/3319535.3354252

[58] STAC. 2017. DARPA space/time analysis for cybersecurity (STAC) program. http://www.darpa.mil/program/space-time-analysis-for-cybersecurity.

[59] François-Xavier Standaert, Tal Malkin, and Moti Yung. 2009. A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks. In *Proceedings of the 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, Vol. 5479. 443–461. https://doi.org/10.1007/978-3-642-01001-9_26

[60] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-Flight Data Load. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (S&P)*. 88–105. https://doi.org/10.1109/SP.2019.00087

[61] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean M. Tullsen, and Deian Stefan. 2021. Automatically eliminating speculative leaks from cryptographic code with blade. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434330

[62] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security* 4, 2/3 (1996), 167–188. https://doi.org/10.3233/JCS-1996-42-304

[63] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2019. oo7: Low-overhead Defense against Spectre Attacks via Program Analysis. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2953709

[64] Jingbo Wang, Chungha Sung, and Chao Wang. 2019. Mitigating power side channels during compilation. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 590–601.

[65] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. 2019. CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 77:1–77:29. https://doi.org/10.1145/3290390

[66] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating Timing Side-Channel Leaks using Program Repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 15–26. https://doi.org/10.1145/3213846.3213851

[67] Meng Wu and Chao Wang. 2019. Abstract Interpretation under Speculative Execution. In *Proceedings of the 40th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 802–815. https://doi.org/10.1145/3314221.3314647

[68] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 428–441. https://doi.org/10.1109/MICRO.2018.00042

[69] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2020. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. *IEEE Micro* 40, 3 (2020), 81–90. https://doi.org/10.1109/MM.2020.2985359

[70] Jun Zhang, Pengfei Gao, Fu Song, and Chao Wang. 2018. SCInfer: Refinement-Based Verification of Software Countermeasures Against Side-Channel Attacks. In *Proceedings of the 30th International Conference on Computer Aided Verification (CAV)*. 157–177.