DeJITLeak: Eliminating JIT-Induced Timing Side-Channel Leaks

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

Julian Andres JiYang
jllandres@shanghaitech.edu.cn
ShanghaiTech University
Shanghai, China

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

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

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

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, DeJITLeak1light, 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], DifFuzz [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.

To foster further research, benchmarks, experimental data and the prototyping tool are released at https://github.com/DeJITLeak.

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 [57]. 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 recompiled 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 [55].

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, 65]). 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 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, 65] 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 compilation.

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 still be 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 [61]. 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.
which may be iteratively recompiled later. Hence we assign to $y$ the version number $V_m$, where the bytecode has the version number $0$. $V_m > 0$ is the highest version number. A directive $d \in D_m$ is valid if $m = d(m)$ and $V_m > V_{m'}$ otherwise it is an invalid directive. Intuitively, the version number $V_m$ indicates the optimized level of the method. JIT recompilation only uses increasingly aggressive optimization techniques, and rolls back to the bytecode version otherwise.

### Syntax

Let $\mathit{LVar}$ (resp. $\mathit{GVar}$) be the finite set of local (resp. global) variables, $\mathit{Val}$ be the set of values, $\mathit{M}$ be a finite set of methods. A program $P$ comprises a set of methods and each method is a list of instructions taken from the instruction set in Figure 3. All these instructions are standard except for $\mathit{deopt}$ which models uncommon traps (cf. Section 2).

For each method $m$, $m[i]$ denotes the instruction at the program point $i$ and $\arg(m)$ 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 $D_m$ be the set of directives of the method $m$, and $d(m)$ the method after JIT compilation according to the directive $d$. In particular, we use $d_0 \in D_m$ 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 $V_m$, where the bytecode has the version number 0, and $V_{m'} > 0$ is the highest version number. A directive $d \in D_m$ is valid if $m = d(m)$ and $V_{m'} > V_{m'}$ otherwise it is an invalid directive. Intuitively, the version number $V_m$ indicates the optimized level of the method. JIT recompilation only uses increasingly aggressive optimization techniques, and rolls back to the bytecode version otherwise.

### State

A state is a tuple $(pc, m, p, os)$, where $pc \in \mathbb{N}$ is the program counter pointing to the next instruction, $m \in \mathit{M}$ is the current executing method, $p : \mathit{LVar} \rightarrow \mathit{Val}$ is a partial function from local variables to values, and $os \in \mathit{Val}^* \mathit{Conf}$ is the operand stack. We denote by $\mathit{States}$ 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 $os_1, os_2 \in \mathit{Val}^*$, let $os_1 \cdot os_2$ denote their concatenation. The empty operand stack is denoted by $\emptyset$.

### Operational semantics

The small-step operational semantics of JVM$_{JIT}$ is given in Figure 2 as a relation $\rightarrow \subseteq \mathit{Conf} \times \mathit{Conf}$, where $\rightarrow \subseteq \mathit{States} \times \mathit{States}$ is an auxiliary relation. Note that the directive $d$ 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 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.

### Instruction set of JVM$_{JIT}$

$\mathit{LVar}$ is a local variable and $y$ is a global variable.
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 $\texttt{goto } j$ unconditionally jumps to program point $j$.

Instruction return ends the execution of the current method and returns the top value $v$ of the 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 $\texttt{deopt m}$ 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 $c')$. 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 $c' \cdot c$ as if the method $m$ were not JIT compiled.

The semantics of invoke $m'$ depends on the directive $d$. If $d = d_{\text{opt}}$, the version of $m'$ in the code heap $ch$ remains the same. If $d$ is valid, i.e., the version number $V_{m'}$ of the optimized version $m' = d(m')$ is larger than that of the current one $V_{m'}, m''$ is stored in the code heap $ch$. After that, it pops up the top $\text{argv}(m')$ 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 $d^*$ for a configuration $c$ is a sequence of valid directives such that the program will not get stuck when starting from $c$ and following $d^*$ for method invocations. A valid schedule $d^*$ yields a JIT-execution $c_0 \downarrow d^* \cdot c_n$ 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 $0 \leq i < n$, either $c_i \rightarrow c_{i+1}$ or $c_i \rightarrow d_i \cdot c_{i+1}$. We require that $d^*$ is equal to the sequence of directives along the JIT-execution, i.e., the concatenation of $d_i$'s. A JIT-free execution is thus a JIT-execution $c_0 \downarrow d^* \cdot c_n$. In this work, we only consider programs that always terminate.

### 3.2 JIT Optimization of JVM

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 $m[i] = \texttt{ifeq } j$. (ifeq is handled accordingly.) Let $B_T$ (resp. $B_I$) be the instructions appearing in the if-(resp. else-)branch of $m[i]$, and the last instruction $m[i']$ of $B_T$ is goto $j'$. The first and last instructions of $B_I$ are $m[j+1]$ and $m[j'-1]$ respectively.

If the profiling data show that the program favors the if-branch, the branch prediction optimization transforms the method $m$ into a new method $m_1$, similar to $m_1$, except that (1) the conditional instruction $\texttt{ifeq } j$ is replaced by $\texttt{ifeq } m[-j+i+1]$ which is immediately followed by the if-branch $B_1$; (2) the else-branch $B_I$ is moved to the end of the method starting at the point $|m|+j+i-1$ and the target point of the last instruction $\texttt{goto } j'$ is revised to $j' - j + i + 1$.

We denote by $T_{bp}(m, i, e-b)$ and $T_{bp}(m, i, i-b)$ 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 $m[i] = \texttt{ifeq } j$ with the if-branch $B_T$ and else-branch $B_I$ (ifeq 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 $T_{bp}(m, i, e-b)$ 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 $T_{oc}(m, i, e-b)$ and $T_{oc}(m, i, i-b)$ 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 \( d \) of a method \( m \) as a pair \((t, \omega)\), where \( t \) is a labeled tree specifying the method invocations to be inlined, and \( \omega \) 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 \((n_1, n_2)\) denoting that the method \( L(n_2) \) is invoked at the call site \( t \) 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 \( \omega \) is of the form \((T_1, i_1, b_1), \ldots, (T_k, i_k, b_k)\), where for every \( 1 \leq j \leq k \), \( T_j \in \{T_{bc}, T_{pc}\} \) 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 \( \omega \), 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 \( c_0 \approx_{pub} c_0' \) if two configurations \( c_0 \) and \( c_0' \) agree on the public input variables, and denote by \( c_0 \approx_{ch} c_0' \) 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 \( d^* \) for \( c_0 \), we have: \( c_0 \parallel_d^* c \) if and only if \( c_0 \parallel_{d^*} c' \).

If the output variables are partitioned into public and secret, we denote by \( c \approx_{pub} c' \) 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 \( c_0 \approx_{pub} c_0' \) and each pair of valid schedules \( d^*_0 \) and \( d^*_0 \) for \( c_0 \) and \( c_0' \) respectively, we have: \( c_0 \parallel d^*_0 \parallel d^*_0 \parallel_{pub} c' \) if \( c_0 \parallel_{d^*_0} c \approx_{pub} c' \) and \( c_0 \parallel_{d^*_0} c \parallel_{d^*_0} c' \). The theorem states that observing public output variables cannot distinguish secret inputs without JIT compilation if 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 \( c_{bc} \) and \( c_{nc} \) be the cost functions for instructions from the bytecode and native code, respectively. We denote by \( c_f(i) \) the cost of the instruction \( i \), which is \( c_{bc}(i) \) if \( i \) is in bytecode mode, otherwise \( c_{nc}(i) \). We lift the function \( c_f \) to states and configurations as usual, e.g., \( c_f((pc, m, \rho, os)) = c_f(m[pc]) \). The cost \( c_f(c_0 \parallel d^*_0 \parallel c_0') \) of a JIT-execution \( c_0 \parallel d^*_0 \parallel c_0' \) is the sum of all the costs of the executed instructions, i.e., \( \sum_{i \in I} c_f(i) \).

**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 \( c_0 \approx_{pub} c_0' \) and the code heaps of \( c_0 \) and \( c_0' \) have the same bytecode instructions, we have: \( c_f(c_0 \parallel d^*_0 \parallel c) = c_f(c_0' \parallel d^*_0 \parallel c') \).

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 \( c_0 \parallel_{d, ch} c_0 \) and a method \( m \). We denote by \( proj_m(c_0 \parallel_{d, ch} c_0) \) the projection of the sequence of executed instructions in \( c_0 \parallel_{d, ch} c_0 \) onto the pairs \((i, m')\) of which consists of a program point \( i \) and a version \( m' \) of the method \( m \). A proper prefix \( \pi \) of \( proj_m(c_0 \parallel_{d, ch} c_0) \) can be seen as the profiling data of the method \( m \). We leave runtime profiling abstract in order to model a large variety of JIT compilations and use \( \pi \) to denote the profiling data of \( m \) after executed instructions \( \pi \) of \( m \) or its compiled versions.

Fix a profiler \( Pf \) that provides one compilation directive \( Pf_m(\pi) \) of a method \( m \) using the profiling data \( \pi \). The schedule \( d^* \) is called a \( Pf\)-schedule if, for each method \( m \) and proper prefix \( \pi \) of \( proj_m(c_0 \parallel_{d, ch} c_0) \), the next compilation directive of \( m \) in \( d^* \) after \( \pi \) is \( Pf_m(\pi) \).

**Lemma 3.4.** For each pair of initial configurations \((c_0, c_0')\) of \( P \) with \( c_0 \approx_{ch} c_0' \) and each pair of valid \( Pf\)-schedules \( d_{\pi, c}^* \) and \( d_{\pi, c'}^* \) for \( c_0 \) and \( c_0' \) respectively, we have: for every method \( m \), every pair \((\pi_1, \pi_2)\) of proper prefixes of \( proj_m(c_0 \parallel_{d, ch} c) \) and \( proj_m(c_0' \parallel_{d, ch} c') \) respectively, if \( \pi_1 \approx_{pub} \pi_2 \) then \( Pf_m(\pi_1) = Pf_m(\pi_2) \).

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 \( c_0 \approx_{ch} c_0' \) and \( c_0' \) respectively, every pair of valid \( Pf\)-schedules \( d_{\pi, c}^* \) and \( d_{\pi, c'}^* \) for \( c_0 \) and \( c_0' \) respectively satisfies \( c_f(c_0 \parallel_{d_{\pi, c}^*} c) = c_f(c_0' \parallel_{d_{\pi, c'}^*} c') \).

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.
The first level of our protection mechanism is to disable JIT compilation and inlining of methods which potentially induce leaks. We denote by PM1 the set of methods that cannot be JIT compiled or inlined, while methods m′ ∈ M \ PM1 can be JIT compiled or inlined. The second level is to disable JIT optimization of branch points in methods M \ PM1 whose optimization will potentially induce leaks. We denote by PM2 the mapping from M \ PM1 to sets of branch points that cannot be JIT optimized. When the method m is JIT compiled, PM2(m) will be updated accordingly.

From the perspective of JVM semantics, the compilation directive of any method from PM1 is limited to d0, and the compilation directives of any method m′ ∈ M \ PM1 can neither inline a method from PM1 nor optimize the branch at a program point in PM2(m′).

A compilation policy of a program P is given by a pair (PM1, PM2). A pt-schedule d* that is compliant to the compilation policy (PM1, PM2) is called a (PM1, PM2)-schedule.

4.2 Type System and Inference

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

4.2.1 Type System

The first level of our protection mechanism is to disable JIT compilation and inlining of methods which potentially induce leaks. The typing rules for inferences rely on its path context pt, which indicates whether an instruction is contained in a secret branch. We use functions ht : GVar → L and lt : LVar → 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 ⊑ is lifted to the functions and the stack type as usual, e.g., ht_1 ⊑ ht_2 if ht_1(y) ⊑ ht_2(y) for each y ∈ GVar.

A typing judgment for non-return instructions is of the form m, i : (pt_1, ht_1, lt_1, st_1) ⇒ (pt_2, ht_2, lt_2, st_2), where m is the method under typing, i is a program point in m. This judgment states that, given the typing context (pt_1, ht_1, lt_1, st_1), the instruction m[i] yields a new typing context (pt_2, ht_2, lt_2, st_2). A typing judgment of the return is of the form m, i : (pt, ht, lt, st) ⇒ (ht, τ), where ht is the security level 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, se_m(i) is a typing context (ht, τ) if m[i] is a return instruction, and (pt, ht, lt, st) otherwise.

Method signature. (A security) signature of a method m is of the form (pt, ht_1, lt_1) ← m (ht_2, τ), which states that, given the typing context (pt, ht_1, lt_1), each global variable y ∈ GVar has the security level 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 ∈ M, m ∈ PM1 if the path context pt in 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.

Figure 6: Typing rules
Rule (T-CALL) ensures that the context of invoke \( m' \) meets the signature \( \text{sig}_P(m') = (p_1, h_1, l_1) \). Let \( \text{pt} \subseteq p_1 \) avoiding that the current path context \( p_t \) has a security level greater than the expected one \( p_1 \), and \( r_0 \subseteq L_1(x_0) \cdot \cdot \cdot r_2 \subseteq L_1(x_2) \) 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 \( \text{nxt}(i) \) denote its junction point, i.e., the immediate post-dominator of \( i \). (Recall that we assumed there is no early return in branches, thus \( \text{junc}(i) \) is well-defined.) We denote by \( \text{region}(i) \) the set of program points \( j \) that can be reached from the branch point \( i \) and are post-dominated by \( \text{junc}(i) \). We denote by \( \text{maxBP}(j) \) the set of branch points \( i \) such that \( j = \text{junc}(i) \) and \( \text{region}(i) \not\subseteq \text{region}(i') \) for any \( i' \in \text{maxBP}(j) \). Intuitively, \( \text{maxBP}(j) \) contains the branch points \( i \) with the junction point \( j \) and \( \text{region}(i) \) is not contained by \( \text{region}(i') \) 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. \( \text{sig}_P \) and \( (PM_1, PM_2) \), denoted by \((PM_1, PM_2, \text{sig}_P) \triangleright m \), if there exists a security environment \( s_m \) such that \( s_m(0) = (pt, h_1, l_1, e) \) for \( \text{sig}_P(m) = (pt, h_1, l_1) \). In the following conditions hold for each program point \( i \):

- If \( i \) is not a junction point, then \( m(i) \triangleright s_m(i) \) implies \( \text{sig}_P(i) \) are the same. This assumption is reasonable in practice, as it allows the inlining of the methods \( m \) if the path context \( pt \) is well-defined.

\[ \text{maxBP}(i) = \{ \text{nxt}(i) \} \cdot \text{next}(i) \]

Intuitively, \((PM_1, PM_2, \text{sig}_P) \triangleright m \) requires that (i) secret branches are forbidden to be optimized by \( PM_2 \) and (ii) methods \( m' \) invoked in \( \text{region}(i) \) of any secret branches \( m[i] \) are forbidden to be JIT compiled and inlined. Recall that we have assumed \( m' \in PM_1 \) if the path context \( pt \) in \( \text{sig}_P(m') \) meets the specified security level \( H \). A program \( P \) is **typable** w.r.t. \( \text{sig}_P \) and \( (PM_1, PM_2) \), denoted by \((PM_1, PM_2, \text{sig}_P) \triangleright P \), if (i) the signature \( \text{sig}_m \) of the entry point \( m \) is \( (L, h_1, l_1) \) such that \( h(y) = H \) and \( l(x) = H \) for any secret inputs \( x, y \) and (ii) \((PM_1, PM_2, \text{sig}_P) \triangleright m \) for every \( m \in M \).

**Theorem 4.1.** If program \( P \) is time-balanced and \((PM_1, PM_2, \text{sig}_P) \triangleright P \), then \( P \) is JIT-time-balanced under \((PM_1, PM_2)\)-schedules.

The proof is provided in the technical report [55]. Note that the native code in the code heap of each initial configuration can only be compiled from bytecode following the policy \((PM_1, PM_2)\).

### 5 Implementation for Practical Java

We have implemented our approach as a tool \text{DeJITLeak} for real-world Java bytecode (jar packages). \text{DeJITLeak} consists of two main components: type inference for computing a signature \( \text{sig}_P \) and a policy \((PM_1, PM_2)\) such that \((PM_1, PM_2, \text{sig}_P) \triangleright P \), and a protection mechanism in HotSpot from OpenJDK [53].

#### 5.1 Type Inference

Our type inference is based 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 \( s_m \) and a signature \( \text{sig}_P(m) \) 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 \( PM_2(m) \), as they can potentially induce \text{TOrFTI} and \text{TBarAN} leaks when JIT optimized.

From the branch points \( PM_2(m) \), we identify and extract all the methods invoked within \( \text{region}(i) \) for all the branch points \( i \in PM_2(m) \). These methods can potentially induce \text{TMETH} leaks when JIT compiled or inlined, thus, are added in \( PM_1 \). According to our type system and the soundness of JOANA, the program \( P \) is typable w.r.t. \( \text{sig}_P \) and \((PM_1, PM_2)\), i.e., \((PM_1, PM_2, \text{sig}_P) \triangleright P \) holds.

#### 5.2 Protection Mechanism in HotSpot

To enforce a compilation policy \((PM_1, PM_2)\) during JIT compilation, we modify HotSpot to demonstrate our approach. To prevent a method \( m \in PM_1 \) 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 \text{exclude} disables JIT compilation of the method \text{signature_of_the_method}, and \text{dontinline} prevents the method \text{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 \text{dontprune} that allows us to specify branch points. The command \text{dontprune} is used similar to \text{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 \text{DeJITLeak\_light}

To reduce performance overhead, we also propose and implement an alternative protection mechanism \text{DeJITLeak\_light}. It only disables the inlining of the methods \( m \in PM_1 \) whereas \text{DeJITLeak\_light} disables both JIT compilation and inlining of the methods \( m \in PM_1 \).

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 DeJITLeakLight 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 tomatc, pac4j, and tourplanner from Themis, as tomatc 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) DeJITLeakLight 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 $T_{\text{taint}}$ (ms) shows the execution time of type inference, column $T_{\text{total}}$ (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, 58]. The mutual information of a program containing a vulnerable conditional statement with the secret condition $K$ and execution time $T$ is defined as $I(K, T) = H(K) - H(K|T)$, where $H(K)$ is classical Shannon entropy measuring uncertainty about $K$, and $H(K|T)$ is the conditional Shannon entropy of $K$ given $T$. $I(K, T)$ 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 $T$.

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 DeJITLeakLight perform significantly better than DisableC2 and MExclude, even better than NOJIT on some benchmarks (e.g., md5, array, login, loopbranch, sanidaydk and jetty); (4) DeJITLeak and DeJITLeakLight 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) DeJITLeakLight brings the least runtime overhead (up to 1.82 times).

On some benchmarks (e.g., authmreloaded, array, login, loopbranch, sanidaydk and jetty), 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,

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>#Node</th>
<th>#Edge</th>
<th>$T_{\text{taint}}$ (ms)</th>
<th>$T_{\text{total}}$ (s)</th>
<th>Mem (Mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td>13</td>
<td>11</td>
<td>20</td>
<td>20.24</td>
<td>1.63</td>
<td>242</td>
</tr>
<tr>
<td>md5</td>
<td>13</td>
<td>46</td>
<td>67</td>
<td>24.47</td>
<td>2.58</td>
<td>249</td>
</tr>
<tr>
<td>salted</td>
<td>13</td>
<td>51</td>
<td>82</td>
<td>26.13</td>
<td>2.59</td>
<td>259</td>
</tr>
<tr>
<td>stringutils</td>
<td>194</td>
<td>15</td>
<td>25</td>
<td>21.52</td>
<td>1.66</td>
<td>244</td>
</tr>
<tr>
<td>authmreloaded</td>
<td>19</td>
<td>43</td>
<td>62</td>
<td>23.91</td>
<td>4.02</td>
<td>396</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>#Node</th>
<th>#Edge</th>
<th>$T_{\text{taint}}$ (ms)</th>
<th>$T_{\text{total}}$ (s)</th>
<th>Mem (Mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>35</td>
<td>2</td>
<td>1</td>
<td>16.21</td>
<td>1.00</td>
<td>164</td>
</tr>
<tr>
<td>gpt14</td>
<td>51</td>
<td>17</td>
<td>22</td>
<td>38.15</td>
<td>2.01</td>
<td>317</td>
</tr>
<tr>
<td>k96</td>
<td>40</td>
<td>21</td>
<td>33</td>
<td>51.43</td>
<td>2.16</td>
<td>331</td>
</tr>
<tr>
<td>login</td>
<td>53</td>
<td>2</td>
<td>1</td>
<td>17.33</td>
<td>0.99</td>
<td>164</td>
</tr>
<tr>
<td>loopbranch</td>
<td>48</td>
<td>2</td>
<td>1</td>
<td>16.55</td>
<td>0.97</td>
<td>160</td>
</tr>
<tr>
<td>modpow1</td>
<td>141</td>
<td>23</td>
<td>35</td>
<td>49.54</td>
<td>2.13</td>
<td>294</td>
</tr>
<tr>
<td>modpow2</td>
<td>106</td>
<td>14</td>
<td>23</td>
<td>31.80</td>
<td>2.00</td>
<td>312</td>
</tr>
<tr>
<td>passwordEq</td>
<td>38</td>
<td>5</td>
<td>6</td>
<td>18.81</td>
<td>1.55</td>
<td>237</td>
</tr>
<tr>
<td>saniday</td>
<td>30</td>
<td>2</td>
<td>1</td>
<td>16.36</td>
<td>0.96</td>
<td>160</td>
</tr>
<tr>
<td>unixlogin</td>
<td>43</td>
<td>21</td>
<td>29</td>
<td>22.65</td>
<td>1.26</td>
<td>193</td>
</tr>
<tr>
<td>bootauth</td>
<td>125</td>
<td>21</td>
<td>34</td>
<td>23.89</td>
<td>3.80</td>
<td>340</td>
</tr>
<tr>
<td>jdk</td>
<td>22</td>
<td>2</td>
<td>1</td>
<td>16.66</td>
<td>0.99</td>
<td>164</td>
</tr>
<tr>
<td>jetty</td>
<td>32</td>
<td>4</td>
<td>5</td>
<td>18.40</td>
<td>1.54</td>
<td>233</td>
</tr>
<tr>
<td>orientdb</td>
<td>211</td>
<td>61</td>
<td>97</td>
<td>39.01</td>
<td>5.52</td>
<td>453</td>
</tr>
<tr>
<td>picterbox</td>
<td>47</td>
<td>6</td>
<td>7</td>
<td>18.57</td>
<td>1.58</td>
<td>238</td>
</tr>
<tr>
<td>spring</td>
<td>39</td>
<td>7</td>
<td>7</td>
<td>19.90</td>
<td>1.75</td>
<td>277</td>
</tr>
</tbody>
</table>
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.

### 6.3 Case Study

We discuss some interesting case studies below.

- **array, login, loopbranch, straightforward, 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 straightforward 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\textsubscript{light} are able to significantly reduce the leakage of array, login, loopbranch, and straightforward. 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-unsuitable 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 FpServer that pads a string to a specified length, where an insecure version would leak information about the original string’s length. DeJITLeak and DeJITLeak\textsubscript{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\textsubscript{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., from\textsuperscript{JSON, get\textit{Time} and get\textit{Expires}}) invoked in secret-dependent branches, but other methods can be JIT compiled 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, 60, 62, 67, 68]. 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, 64, 65], 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, 56, 59]. This problem has been recently studied [21, 26, 37–39, 41, 60, 62, 66–68], where speculative execution semantics, notions of constant-time under the new semantics, detection and mitigation approaches, etc, have been proposed. Among them, Blade [60] 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, 63, 69] 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 fine-grained 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 DeJITLeak1light 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.

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


