Preventing Timing Side-Channels via Security-Aware Just-In-Time Compilation

Qi Qin
ShanghaiTech University
Shanghai, China

JulianAndres JiYang
ShanghaiTech University
Shanghai, China

Fu Song
ShanghaiTech University
Shanghai, China

Taolue Chen
Birkbeck, University of London
London, United Kingdom

Xinyu Xing
Northwestern University
Evanston, USA

Abstract

Recent work has shown that Just-In-Time (JIT) compilation can introduce timing side-channels to constant-time programs, which would otherwise be a principled and effective means to counter timing attacks. In this paper, we propose a novel approach to eliminate JIT-induced leaks from these programs. Specifically, we present an operational semantics and a formal definition of constant-time programs under JIT compilation, laying the foundation for reasoning about programs with JIT compilation. We then propose to eliminate JIT-induced leaks via a fine-grained JIT compilation for which we provide an automated approach to generate policies and a novel type system to show its soundness. We develop a tool DeJITLeak for Java based on our approach and implement the fine-grained JIT compilation in HotSpot. Experimental results show that DeJITLeak can effectively and efficiently eliminate JIT-induced leaks on three datasets used in side-channel detection.

1 Introduction

Timing side-channel attacks allow an adversary to infer secret information by measuring the execution time of an implementation, thus pose a serious threat to secure systems [33]. One notorious example is Lucky 13 attack that can remotely recover plaintext from the CBC-mode encryption in TLS due to an unbalanced branch statement [3].

Constant-time principle, which requires the execution time of an implementation being independent of secrets, is an effective countermeasure to prevent such attacks. However, writing constant-time programs is error-prone. For instance, even though two protections against Lucky 13 were implemented in AWS's s2n library, a variant of Lucky 13 can remotely and completely recover plaintext from the CBC-mode cipher suites in s2n [2]. Therefore, various approaches have been proposed for automatically verifying constant-time security of high-/intermediate-level programs, e.g., ct-verif [4] and CacheAudit [22] for C programs, CT-Wasm [52] for WASM programs, Blazer [5] and Themis [20] for Java programs, and FaCT [19] for eliminating leakages.

However, constant-time programs may be still vulnerable in practice if the runtime environment is not fully captured by constant-time models. For instance, static compilation from high-/intermediate-level programs to low-level counterparts can destruct constant-time security [8–10]; constant-time executable programs are vulnerable in modern processors due to, e.g., speculative or out-of-order execution [18, 32, 36]; JIT compilation makes constant-time bytecode vulnerable [14, 16], called JIT-induced leaks hereafter. In this work, we focus on JIT-induced leaks.

JIT compilation has been used in numerous programming language engines (e.g., PyPy for Python, LuaJIT for Lua, HotSpot for Java, and V8 for JavaScript) to improve performance. However, JIT compilation can break the balance of conditional statements, e.g., some methods are JIT compiled or inlined in one branch but not in the other branch, or one branch is speculatively optimized, making constant-time programs vulnerable at runtime as shown in [14, 16]. Despite the serious risk of JIT compilation, there is no rigorous approach to eliminate JIT-induced leaks otherwise completely disabling JIT compilation.

In this work, we aim to automatically and rigorously eliminate JIT-induced leaks. Our contributions are both theoretical and practical. On the theoretical side, we first lay the foundations for timing side-channel security under JIT compilation by presenting a formal operational semantics and defining a notion of constant-time for a fragment of the JVM under JIT compilation. We do not model concrete JIT compilation as done by [6, 24]. Instead, we leave them abstract in our model and model JIT compilation via compilation directives controlled by the adversary. This allows to consider very powerful attackers who have control over JIT compilation. It also makes it possible to reason 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 checking the effectiveness of policies for fine-grained JIT compilation.

PL'18, January 01–03, 2018, New York, NY, USA
2018.

On the practical side, we propose DeJITLeak, an automatic technique to generate policies 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 implement DeJITLeak as a tool and fine-grained JIT compilation in HotSpot JVM from OpenJDK. We conduct extensive experiments on three datasets used in recent side-channel detection: DifFuzz [41], Blazer [5] and Themis [20]. Experimental results show that our approach significantly outperforms the strategies proposed in [13]. We report interesting case studies which shed light on directions for further research in this area.

In summary, our contributions are:

- A formal treatment of JIT-induced leaks including an operational semantics and a constant-time notion under JIT compilation;
- A protection mechanism against JIT-induced leaks via a fine-grained JIT compilation and an efficient approach to generate policy 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.

2 Overview

In this section, we first give a brief overview of the side-channel leaks induced by JIT compilation [14]. We will exemplify these JIT-induced leaks using the HotSpot virtual machine (HotSpot for short) on OpenJDK 1.8. We then give an overview of our approach to identify and eliminate the JIT-induced leaks automatically.

2.1 JIT-Induced Leaks

JIT-induced leaks could be caused at least by the following three JIT compilation techniques, i.e., (1) optimistic compilation, (2) branch prediction, and (3) method compilation.

Optimistic compilation (Topti). Optimistic compilation is a type of speculation optimizations [6]. During the JIT compilation of a method, the compiler speculates on the most likely executed branches by pruning rarely executed branches. Therefore, 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 (known as an uncommon trap in HotSpot) 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 a longer time to deoptimize and roll back to the bytecode. This difference in execution time induces the Topti timing side-channel even if branches are balanced in bytecode. When the attacker can feed inputs to the program, Topti could be triggered for a conditional statement whose condition relies on secrets. The attacker would then be able to infer the secret information from the difference between execution time.

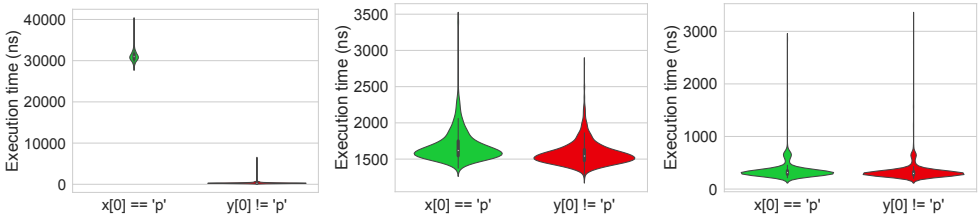
As an example, consider the pwdEq method shown in Figure 1a, which is extracted and simplified from the DARPA Space/Time Analysis for Cybersecurity (STAC) engagement program *gabfeed_1* [46]. It takes strings a and b with length 8 as inputs, denoting the user-entered and correct passwords, respectively. It checks if they are identical within a loop. The flag equal is assigned by false if two chars mismatch. To balance execution time, the dummy flag shmequal is introduced and assigned by false if two chars match.

The pwdEq method is marked as safe in STAC and would be verified as safe by the timing side-channel verification tools Blazer [5] and Themis [20] which do not consider JIT compilation. However, it indeed 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 produce this, we use two random 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 by executing pwdEq with inputs (x,"password") and (y,"password"), respectively. This mimics the process that an attacker guesses the secret data char-by-char, avoiding guessing the entire secret data simultaneously. The distribution of the execution time is shown in Figure 1b. As a cross reference, 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 the attacker to infer if the first char is correctly guessed.

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 spacial locality of instruction cache. However, the reordering of basic blocks unbalances the execution time of branches even if it is balanced in bytecode.

If the attacker can feed inputs to the program, Tbran could be triggered for a conditional statement whose condition relies on secrets, and thus the attacker could be able to infer secret information by measuring the execution time. Although the difference in the execution time between branches via

```
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

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. In practice, a sophisticated multi-tier compilation mechanism is adopted (in e.g., HotSpot), where a method may be recompiled multiple times to more optimized native code to further improve the performance. Meanwhile, during compilation, frequently invoked small methods could be inlined to speed up execution.

For a conditional statement with method invocations, if the 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 induces a timing side-channel, called Tmeth.

Concrete demonstrations of Tbran and Tmeth are given in the supplementary material.

2.2 Automatically Eliminating JIT-induced Leaks

In this work, we present an automated, rigorous approach to eliminate the above-mentioned JIT leaks. We do not consider CPU-level speculative leaks [18, 32] and cache-induced leaks [4, 23] for which various mitigation techniques have been proposed in literature, e.g., [30, 49, 51, 53, 55, 56]. We assume that the bytecode program is of constant-time, which can largely be achieved by existing work (e.g., [1, 5, 19, 20, 39]). Our goal is to protect constant-time bytecode programs from the JIT-induced leaks discussed before. In general, there are trace-based and method-based JIT compilation approaches [31], and we shall focus on the latter in this work.

A straightforward way to prevent JIT-induced leaks is to simply disable JIT compilation completely or JIT compilation of the chosen methods. Indeed, [13] proposed the following three compilation strategies, named NOJIT, DisableC2 and MExclude. (1) 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. (2) 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. (3) The MExclude strategy disables JIT compilation for the user-chosen methods instead of the entire program, by which some leaks (i.e., Tbran and Topti) can be prevented. This strategy cannot prevent from Tmeth, but its main shortcoming is that there is no protection for non-chosen methods, and it is also unclear how to choose methods to disable. In summary, the existing compilation strategies either incur a high performance cost or fail to prevent all the known JIT-induced leaks.

Disabling JIT compilation at the method level is indeed unnecessary. Essentially, we only need to ensure that secret information will not be leaked when the methods are JIT compiled or inlined. An important observation is 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;

Based on the above observation, we propose a novel approach DeJITLeak, to 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 compilation strategies proposed in [13].

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 [50]. 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.

We introduce a type system to prove the soundness of the fine-grained JIT compilation, i.e., under which the resulting program is free of the aforementioned JIT-induced leaks if its bytecode version is leakage-free. To this end, we introduce a JVM submachine and formulate an operational semantics with JIT compilation. We also provide a notion of JIT-constant-time to formalize timing side-channel security of programs under JIT compilation. We show that a constant-time program remains constant-time 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.

2.3 Threat Model

In this work, we focus on timing side-channel leaks induced by branch prediction, optimistic compilation and method compilation. We assume that the adversary is able to influence how bytecode is JIT compiled and deoptimized by feeding inputs to programs to trigger branch prediction and optimistic compilation of chosen conditional statements, or method compilation and deoptimization of chosen methods. The time for JVM profiling, JIT compilation and garbage collection is not taken into account, as they are often performed in distinct threads. We do not consider other JIT optimizations such as constant propagation, loop unfolding and dead elimination, which are often difficult to be controlled by the adversary at runtime. To the best of our knowledge, no existing attack leverages these optimizations. We do not consider CPU-level optimizations (such as speculative execution and cache) which have been studied, e.g., [18, 30, 49, 51, 55, 56].

3 The Language: Syntax, Semantics and Constant-Time

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

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. Both bytecode and native code are presented in JVM_{JIT}. Note that this is for the sake of presentation, as 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, each of which is a list of instructions taken from the instruction set in Figure 2. All these instructions are standard except for the instruction deopt `md` which is used to model uncommon traps (cf. Section 2) .

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 2. Instruction set of JVM_{JIT}, where $x \in$ **LVar** is a local variable and $y \in$ **GVar** is a global variable

For each method m, $m[i]$ denotes the instruction in m 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. We denote by $\mathbf{D}_m$ the set of directives of the method m, and by $\mathbf{d}(m)$ the resulting version after compilation and optimization 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 Section 3.2.3.

In general, a method in bytecode is compiled into native code which may be iteratively recompiled later. Thus, we assign to each method m a version number $\mathcal{V}_m$, where the bytecode has the version number 0, and the highest version number is $\mathcal{V}_{\max} > 0$. A directive $\mathbf{d} \in \mathbf{D}_m$ is invalid 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 and configuration. A state is a tuple $\langle \text{pc}, m, \rho, \text{os}\rangle$ where

- $\text{pc} \in \mathbb{N}$ is the program counter that points to the next instruction in m;
- $m \in \mathbf{M}$ is the current executing method;
- $\rho : \mathbf{LVar} \to \mathbf{Val}$ is a partial function from local variables to values;
- $\text{os} \in \mathbf{Val}^*$ is the operand stack.

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

$$\frac{m[\mathrm{pc}] = \mathrm{ifeq}\ j \quad v = 0}{\langle \mathrm{pc}, m, \rho, v \cdot \mathrm{os}\rangle \rightsquigarrow \langle j, m, \rho, \mathrm{os}\rangle} \qquad \frac{m[\mathrm{pc}] = \mathrm{ifeq}\ j \quad v \neq 0}{\langle \mathrm{pc}, m, \rho, v \cdot \mathrm{os}\rangle \rightsquigarrow \langle \mathrm{pc}+1, m, \rho, \mathrm{os}\rangle} \qquad \frac{m[\mathrm{pc}] = \mathrm{swap}}{\langle \mathrm{pc}, m, \rho, v_1 \cdot v_2 \cdot \mathrm{os}\rangle \rightsquigarrow \langle \mathrm{pc}+1, \rho, v_2 \cdot v_1 \cdot \mathrm{os}\rangle}$$

$$\frac{m[\mathrm{pc}] = \mathrm{ifneq}\ j \quad v \neq 0}{\langle \mathrm{pc}, m, \rho, v \cdot \mathrm{os}\rangle \rightsquigarrow \langle j, m, \rho, \mathrm{os}\rangle} \qquad \frac{m[\mathrm{pc}] = \mathrm{ifneq}\ j \quad v = 0}{\langle \mathrm{pc}, m, \rho, v \cdot \mathrm{os}\rangle \rightsquigarrow \langle \mathrm{pc}+1, m, \rho, \mathrm{os}\rangle} \qquad \frac{m[\mathrm{pc}] = \mathrm{store}\ x \quad x \in \mathrm{dom}(\rho)}{\langle \mathrm{pc}, m, \rho, v \cdot \mathrm{os}\rangle \rightsquigarrow \langle \mathrm{pc}+1, m, \rho[x \mapsto v], \mathrm{os}\rangle}$$

$$\frac{m[\mathrm{pc}] = \mathrm{load}\ x}{\langle \mathrm{pc}, m, \rho, \mathrm{os}\rangle \rightsquigarrow \langle \mathrm{pc}+1, m, \rho, \rho(x) \cdot \mathrm{os}\rangle} \qquad \frac{m[\mathrm{pc}] = \mathrm{goto}\ j}{\langle \mathrm{pc}, m, \rho, \mathrm{os}\rangle \rightsquigarrow \langle j, m, \rho, \mathrm{os}\rangle} \qquad \frac{s \rightsquigarrow s'}{(\mathsf{ch}, \mathsf{h}, s, \mathsf{cs}) \rightarrow (\mathsf{ch}, \mathsf{h}, s', \mathsf{cs})}$$

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

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

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

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

Figure 3. Operational semantics of $\mathrm{JVM}_{\mathrm{JIT}}$, where $\mathrm{dom}(\rho)$ denotes the domain of the partial function ρ

We denote by **States** the set of states. For each function $f : X \to V$, variable $x \in X$ and value $v \in V$, let $f[x \mapsto v]$ be the function where for every $x' \in X$, $f[x \mapsto v](x') = f(x')$ if $x' \neq x$, and $f[x \mapsto v](x') = v$ otherwise. For two operand stacks $\mathrm{os}_1, \mathrm{os}_2 \in \mathbf{Val}^*$, let $\mathrm{os}_1 \cdot \mathrm{os}_2$ denote their concatenation. The empty operand stack is denoted by ϵ.

A configuration is of the form $(\mathsf{ch}, \mathsf{h}, s, \mathsf{cs})$ or (h, v), where ch is a code heap storing the latest version of each method; $\mathsf{h} : \mathbf{GVar} \to \mathbf{Val}$ is a (data) heap, i.e., a partial function from global variables to values; $s \in \mathbf{States}$ is the current state; $\mathsf{cs} \in \mathbf{States}^*$ 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 $(\mathsf{ch}, \mathsf{h}, \langle \mathrm{pc}, m, \rho, \mathrm{os}\rangle, \mathsf{cs})$ is an initial one if $\mathrm{pc} = 0$, m is the entry point of the program, and $\mathrm{os} = \mathsf{cs} = \epsilon$. Let **Conf** denote the set of configurations, $\mathsf{cs}_1 \cdot \mathsf{cs}_2$ be the concatenation of two call stacks cs_1 and cs_2, and ϵ be the empty call stack.

Operational semantics with JIT Compilation. The small-step operational semantics of $\mathrm{JVM}_{\mathrm{JIT}}$ is given in Figure 3 as a relation $\rightarrow \subseteq \mathbf{Conf} \times \mathbf{Conf}$, where $\rightsquigarrow \subseteq \mathbf{States} \times \mathbf{States}$ is an auxiliary relation. Directives $\mathbf{d}$ apply to method invocations only, thus are associated to the relation $\rightarrow$ only for method invocations. The semantics of each instruction is mostly standard except for the method invocation and deoptimization. We only explain some selected ones. Full explanation refers to the supplementary material.

Instruction return ends the execution of the current method, returns the top value v of the current operand stack, either by pushing it on top of the operand stack of the caller and re-executes the caller from the return site if the current method is not the entry point, or enters a final configuration (h, v) if the current method is the entry point.

Instruction deopt md deoptimizes the current 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 $\mathrm{base\_version}(m)$ of the method m is restored into the code heap ch. We assume that 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 method invocation invoke m' depends on the directive $\mathbf{d}$. If $\mathbf{d}$ is $\mathbf{d}_\emptyset$ then the instructions of m' in the code heap ch remain the same. If $\mathbf{d}$ is valid, namely, the optimized version $\mathcal{V}_{m''}$ after applying $\mathbf{d}$ has larger version number than that of the current version $\mathcal{V}_{m'}$, the new optimized version $m'' = \mathbf{d}(m')$ is stored in the code heap ch. After that, it pops the top $|\mathrm{argv}(m')|$ values from the current operand stack, passes them to the formal arguments $\mathrm{argv}(m'')$ of 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. The valid schedule $\mathbf{d}^\star$ yields a JIT-execution, denoted by $c_0 \Downarrow_{\mathbf{d}^\star} c_n$, which is a sequence $c_0 c_1 \cdots c_n$, 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 $c_i \rightarrow_{\mathbf{d}_i} c_{i+1}$. 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 $c_0 \Downarrow_{\mathbf{d}_\emptyset^\star} c_n$. Note that in this

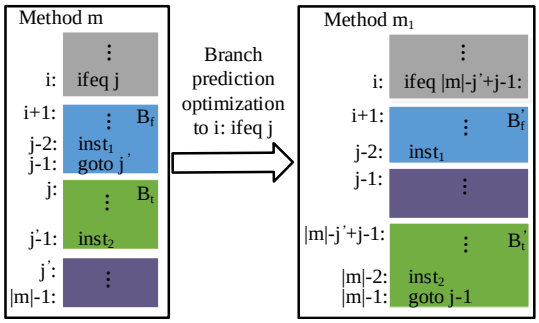


Figure 4. Branch prediction optimization

work, we assume that the execution of a program always terminates.

In the rest of this work, we assume that each method has one return instruction which does not appear in any branch of conditional statements, as early return often introduces timing side-channel leaks.

3.2 JIT Optimization of JVM_{JIT}

In this section, we first introduce branch prediction and optimistic compilation, then define method compilation as well as compilation directives in detail.

3.2.1 Branch prediction. Consider a method m and a conditional instruction $m[i] = \text{ifeq } j$. (We take ifeq as the example, and ifneq can be 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 as shown in Figure 4 for $m[i] = \text{ifeq } j$. The formal definition and an illustrating example are given in the supplementary material.

If the profiling data shows that the program favors the if-branch, the branch prediction optimization to the conditional instruction $m[i] = \text{ifeq } j$ 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 $\mathsf{T}_{\text{bp}}(m, i, \text{else-b})$ and $\mathsf{T}_{\text{bp}}(m, i, \text{if-b})$ the new methods m_1 and m_2 respectively. It is easy to see that the branch prediction optimization transforms the original program to a semantically equivalent program.

3.2.2 Optimistic compilation. Again, consider the conditional instruction $m[i] = \text{ifeq } j$ 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 almost never gets executed, the optimistic compilation optimization transforms the method m into a new method m_1 in a similar way to $\mathsf{T}_{\text{bp}}(m, i, \text{else-b})$. Here, the if-branch B_t is replaced by an uncommon trap. The method m_2 is defined similarly if the profiling data show that the else-branch almost never gets executed. More details refer to the supplementary material.

We denote by $\mathsf{T}_{\text{oc}}(m, i, \text{else-b})$ and $\mathsf{T}_{\text{oc}}(m, i, \text{if-b})$ 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.

3.2.3 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 $n \in V$ 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 $1 \le j \le k$, $\mathsf{T}_j \in \{\mathsf{T}_{\text{bp}}, \mathsf{T}_{\text{oc}}\}$ denotes the optimization type 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.

Note that in our formalism, the optimistic compilation optimization adds one uncommon trap for each conditional statement. In practice, multiple conditional statements may share one uncommon trap, which is not modeled here but can be handled by our approach as well.

3.3 Consistency and Constant-Time

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 $c_0 \simeq_{\text{pub}} c_0'$ if the initial configurations c_0 and c_0' agree on the public input variables, and denote by $c_0 \simeq_{\text{ch}} c_0'$ if c_0 and c_0' have the same code heap.

Consistency. The following theorem 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:*

$$c_0 \Downarrow_{\mathbf{d}^\star_\emptyset} c \text{ iff } c_0 \Downarrow_{\mathbf{d}^\star} c'.$$

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

Constant-time. 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 assume that, for each pair $(\mathsf{inst}_1, \mathsf{inst}_2)$ of instructions, $\mathsf{cf}_{\mathsf{bc}}(\mathsf{inst}_1) = \mathsf{cf}_{\mathsf{bc}}(\mathsf{inst}_2)$ implies that $\mathsf{cf}_{\mathsf{nc}}(\mathsf{inst}_1) = \mathsf{cf}_{\mathsf{nc}}(\mathsf{inst}_2)$. Namely, the cost equivalence of bytecode instructions are preserved in native code. 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., $\sum_{i=0}^{n-1} \mathsf{cf}(c_i)$.

A program P is constant-time (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 constant-time policy 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-constant-time. To define constant-time under JIT compilation, called JIT-constant-time, we first introduce some notations.

Consider a JIT-execution $c_0 \Downarrow_{\mathbf{d}^\star} c_n$ and a method m, let $\mathsf{proj}_m(c_0 \Downarrow_{\mathbf{d}^\star} c_n)$ denote 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.

Let us fix a profiler pf, which 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 3.3. *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 $pf_m(\widehat{\pi_1}) = pf_m(\widehat{\pi_2})$.

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

A program P is JIT-constant-time 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-constant-time policy requires 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 timing side-channel leaks even if the JIT compilation is enabled. We allow the code heaps in c_0 and c_0' to be mixed with bytecode and native code, because the adversary can run the program multiple times with chosen inputs before launching attacks.

We remark that our definition of pf-schedules considers a powerful adversary who controls executing instructions and thus the compilation directives of methods, 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.

As argued above, a constant-time program P may not be of JIT-constant-time due to JIT compilation. This work aims to prescribe a fine-grained JIT compilation under which a constant-time program P is still JIT-constant-time so there is no need to disable JIT compilation completely.

4 Protect Mechanism and Type System

In this section, we first propose a two-level protection mechanism to eliminate JIT-induced leaks and then present an information-flow type system for proving JIT-constant-time 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 $\mathtt{prot}_1$ the set of methods that cannot be JIT compiled or inlined, i.e., these methods can

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

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

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

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

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

Figure 5. Typing rules

only be executed in the interpreted mode. The second level is to disable JIT optimization of branch points in methods $\mathbf{M} \setminus \texttt{prot}_1$, whose optimization will potentially induce leaks. We denote by $\texttt{prot}_2$ the mapping from $\mathbf{M} \setminus \texttt{prot}_1$ to sets of branch points that cannot be JIT optimized. When the method m is compiled, $\texttt{prot}_2(m)$ will be updated accordingly.

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

For a given program P, a policy for fine-grained JIT compilation is given by a pair $(\texttt{prot}_1, \texttt{prot}_2)$. A pf-schedule $\mathbf{d}^\star$ that is compliant to the policy $(\texttt{prot}_1, \texttt{prot}_2)$ is called a $(\texttt{prot}_1, \texttt{prot}_2)$-schedule.

4.2 Type System and Inference

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

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 can indicates whether an instruction is contained in a secret branch. We use functions $\text{ht} : \mathbf{GVar} \rightarrow \mathbb{L}$ and $\text{lt} : \mathbf{LVar} \rightarrow \mathbb{L}$ 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., $\text{ht}_1 \sqsubseteq \text{ht}_2$ if $\text{ht}_1(y) \sqsubseteq \text{ht}_2(y)$ for each $y \in \mathbf{GVar}$.

The typing judgment for non-return instructions is of the form $m, i \vdash (\text{pt}_1, \text{ht}_1, \text{lt}_1, \text{st}_1) \Rightarrow (\text{pt}_2, \text{ht}_2, \text{lt}_2, \text{st}_2)$, where m is the method under typing, i is a program point in m. This judgment states that, given the typing context $(\text{pt}_1, \text{ht}_1, \text{lt}_1, \text{st}_1)$, the instruction $m[i]$ yields a new typing context $(\text{pt}_2, \text{ht}_2, \text{lt}_2, \text{st}_2)$. The typing judgment of the return is of the form $m, i \vdash (\text{pt}, \text{ht}, \text{lt}, \text{st}) \Rightarrow (\text{ht}, \tau)$, 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, $\text{se}_m(i)$ is a typing context (ht, τ) if $m[i]$ is a return instruction, and $(\text{pt}, \text{ht}, \text{lt}, \text{st})$ otherwise.

Method signature. A (security) signature of a method m is of the form $(\text{pt}, \text{ht}_1, \text{lt}_1) \hookrightarrow_m (\text{ht}_2, \tau)$, which states that, given the typing context $(\text{pt}, \text{ht}_1, \text{lt}_1)$, each global variable $y \in \mathbf{GVar}$ has the security level $\text{ht}_2(y)$ 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 $m \in \mathbf{M}$, $m \in \texttt{prot}_1$ if the path context pt in $\text{sig}_P(m)$ is the high security level H.

Typing rules. The typing rules are presented in Figure 5, where the key premises are highlighted and $(\text{ht}, \tau) \models \text{sig}_P(m)$ means that $\text{ht} \sqsubseteq \text{ht}'$ and $\tau \sqsubseteq \tau'$ for the signature $\text{sig}_P(m) = (\text{pt}, \text{ht}_1, \text{lt}_1) \hookrightarrow_m (\text{ht}', \tau')$.

The type system only checks bytecode programs, thus there is no typing rule for the deoptimization instruction deopt md. Most typing rules are standard. For example, (T-Push), (T-Pop), (T-Bop) and (T-Swap) track the flow of the secret data via the operand stack, including explicit and implicit flows. Similarly, (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 $i \in \mathtt{prot}_2(m)$ if pt′ has the high security level H, otherwise the branches may become unbalanced, resulting in JIT-induced leaks.

Rule (T-Ret) requires $(\mathsf{ht}, \tau) \models \mathsf{sig}_P(m)$ 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 $\mathsf{sig}_P(m)$.

Rule (T-Call) ensures that the context of invoke m' meets the signature $\mathsf{sig}_P(m') = (\mathsf{pt}_1, \mathsf{ht}_1, \mathsf{lt}_1) \hookrightarrow_{m'} (\mathsf{ht}_2, \tau)$, 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 security of a constant-time program under JIT compilation is verified by type inference. To formalize this, we first introduce some notations [11].

Consider 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.

A method m is typable w.r.t. the signature sig_P and policy $(\mathtt{prot}_1, \mathtt{prot}_2)$, denoted by $(\mathtt{prot}_1, \mathtt{prot}_2, \mathsf{sig}_P) \triangleright m$, if there exists a security environment se_m for 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, $(\mathtt{prot}_1, \mathtt{prot}_2, \mathsf{sig}_P) \triangleright m$ requires that (1) secret branches are forbidden to be optimized by $\mathtt{prot}_2$ and (2) 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 $m' \in \mathtt{prot}_1$ if the path context pt in $\mathsf{sig}_P(m')$ has the high security level H.

A program P is typable w.r.t. the signature sig_P and policy $(\mathtt{prot}_1, \mathtt{prot}_2)$, denoted by $(\mathtt{prot}_1, \mathtt{prot}_2, \mathsf{sig}_P) \triangleright P$, if (1) for the entry point m: $\mathsf{sig}_m = (\mathsf{L}, \mathsf{ht}, \mathsf{lt}) \hookrightarrow_m (\mathsf{ht}', \tau)$, $\mathsf{ht}(y) = \mathsf{H}$ and $\mathsf{lt}(x) = \mathsf{H}$ for any secret inputs x, y; and (2) $(\mathtt{prot}_1, \mathtt{prot}_2, \mathsf{sig}_P) \triangleright m$ for every method $m \in \mathbf{M}$.

Theorem 4.1. *Given a program P, if P is constant-time and $(\mathtt{prot}_1, \mathtt{prot}_2, sig_P) \triangleright P$, then P is JIT-constant-time under $(\mathtt{prot}_1, \mathtt{prot}_2)$-schedules.*

The proof is provided in the supplementary material. Note that the native code in the code heap of each initial configuration can only be complied from bytecode following the policy $(\mathtt{prot}_1, \mathtt{prot}_2)$.

5 Implementation

We have implemented the detection and elimination approach as a tool DeJITLeak for real-world Java bytecode (in the form of Jar packages). DeJITLeak consists of two main components: type inference for computing a signature sig_P and a policy $(\mathtt{prot}_1, \mathtt{prot}_2)$ such that $(\mathtt{prot}_1, \mathtt{prot}_2, \mathsf{sig}_P) \triangleright P$, and a modified version of HotSpot from OpenJDK [43] implementing the protection mechanism.

5.1 Type Inference

Our type inference is built on JOANA [29], a sound, flow-, context-, and object-sensitive information flow framework based on program dependence graphs.

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. With se_m and signature $\mathsf{sig}_P(m)$, we can 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 $\mathtt{prot}_2(m)$, as they can potentially induce Topti and Tbran leakage when optimized.

From the branch points $\mathtt{prot}_2(m)$, we identify and extract all the methods invoked within $\mathsf{region}(i)$ for all the branch points $i \in \mathtt{prot}_2(m)$. These methods can potentially induce Tmeth leakage when JIT compiled or inlined. Thus, these methods are added in $\mathtt{prot}_1$. According to our type system, the soundness and precision of our type inference inherit from that of JOANA, namely, the program P is typable w.r.t. the signature sig_P and policy $(\mathtt{prot}_1, \mathtt{prot}_2)$, i.e., $(\mathtt{prot}_1, \mathtt{prot}_2, \mathsf{sig}_P) \triangleright P$ holds.

5.2 Protection Mechanism in HotSpot

To enforce the policy $(\mathtt{prot}_1, \mathtt{prot}_2)$ during JIT compilation, we modify HotSpot from OpenJDK to demonstrate our approach. To prevent a method $m \in \mathtt{prot}_1$ from being compiled and inlined, we use the option `CompileCommand` supported by HotSpot [42], 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 the method is recompiled. We plan to create a pull request of our modification to OpenJDK.

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

In the experiments, we found that disabling JIT compilation of all the methods invoked in secret branches may degrade the performance significantly. To compensate, we propose and implement an alternative protection mechanism DeJITLeak$_{\texttt{light}}$.

DeJITLeak$_{\texttt{light}}$ only disables the inlining of the methods $m \in \texttt{prot}_1$ whereas DeJITLeak disables both JIT compilation and inlining of the methods $m \in \texttt{prot}_1$. 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 strategy for developers to implement a constant-time program by invoking same methods in both sides of each secret branch point.

We remark that inlining method should be disabled even if this method is invoked on both sides of a secret branch point, as the method may be inlined only in one branch, inducing branch unbalance and subsequent leakages.

6 Evaluation

In this section, we report the evaluation of DeJITLeak and DeJITLeak$_{\texttt{light}}$. We first evaluate the efficiency of the type inference, and then compare our protection approach with other strategies: NOJIT, DisableC2, and MExclude (cf. Section 2.2). According to [13], we disable JIT compilation of the methods that contain some secret branch points for MExclude, but methods invoked in secret branches could be JIT complied or inlined.

We conduct experiments on the benchmarks that have been used to evaluate DifFuzz [41], Blazer [5] and Themis [20], including real-world programs from well-known Java applications such as Apache FtpServer, micro-benchmarks from DARPA STAC and classic examples from the literature [25, 33, 44]. Recall that we target constant-time Java bytecode. Thus, we only consider the safe versions of the benchmarks, i.e., programs that are leakage-free or only have slight leaks under their leakage models without the JIT compilation. We also exclude the benchmarks tomcat, pac4j, and tourplanner from Themis, as tomcat and pac4j have significant leakages [16] 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 [21]. 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.

Table 1. Results of type checking

	Name	♯LOC	Time (s)	Memory (Mb)
DifFuzz	clear	38	1.50	307
	md5	65	1.53	305
	salted	82	1.59	309
	stringutils	20	1.69	146
	authmreloaded	76	0.95	246
Blazer	array	17	0.98	244
	gpt14	12	2.18	208
	k96	24	2.03	220
	login	18	0.92	239
	loopbranch	23	0.92	232
	modpow1	22	2.00	227
	modpow2	14	1.95	198
	passwordEq	18	1.57	129
	sanity	15	0.90	236
	straightline	13	1.04	241
	unixlogin	36	1.19	273
Themis	bootauth	112	3.68	402
	jdk	13	0.92	233
	jetty	14	1.51	306
	orientdb	198	5.65	539
	picketbox	39	1.51	130
	spring	25	1.80	150

All experiments are conducted on an Intel NUC running Ubuntu 18.04 with Intel Core I5-8259U CPU @ 2.30GHz and 16GB of memory, without disabling CPU-level and other JIT optimizations when JIT compilation is enabled.

In summary, the results show that (1) DeJITLeak is very effective and is able to successfully eliminate a majority of the leakages induced by JIT compilation, 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 time and memory used for type inference of the benchmarks. We observe that these benchmarks can be solved efficiently. It takes 1.73 seconds on average (up to 5.65 seconds) and 251 Mb to analyze one benchmark. Note that the time and memory consumption does not necessarily correlate with the size of the program (e.g., on *gpt14* vs. *k96*).

Table 2. Evaluation results of DeJITLeak

	Benchmark			NOJIT		DisableC2		MExclude		DeJITLeak		DeJITLeak$_{light}$	
	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**

6.2 Effectiveness and Efficiency of the Protection

Our approach provides the security guarantees of JIT-constant-time w.r.t. the JIT-induced leaks of Topti, Tbran and Tmeth, but there are other JIT and CPU-level optimizations that may induce timing side-channel leaks as well. Thus, we evaluate the effectiveness by quantifying the amount of leakages in practice using mutual information [37], a widely used metric for side channel analysis [34, 35, 38, 47].

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 the attacker has learned the execution time T. We manually create attacks to explore the maximum amount of leakages according to [14]. 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 summarized in Table 2, which records 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 become 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$_{light}$ 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$_{light}$ are almost comparable.

Efficiency. We measure the efficiency of respective method 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; (4) DeJITLeak$_{light}$ 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, MExclude performs worse than DeJITLeak.

We shall discuss some interesting case studies below.

array, login, loopbranch, straightline, unixlogin: Experimental results show that their leakages are significant in practice, although these benchmarks were claimed of leakage-free or only have slight leaks under their leakage models without JIT compilation [5, 20, 41]. 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 [33]. 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 statement in bytecode. Since DeJITLeak only disables JIT compilation of the methods invoked in secret branches, but other methods can be JIT complied including the C2 mode compilation. Thus, DeJITLeak is only able to eliminate the Topti leak, but amplifies the timing difference of these branches compared over the entire execution time, whereas the others do not amplify the timing difference.

7 Related Work

Timing side-channel attacks have attracted many attentions, with a significant amount of work devoted to its detection [15, 38, 41, 44], verification [4, 5, 7, 12, 20, 22, 23] and mitigation [1, 19, 22, 39, 52, 53], which vary in targeted programs, leakage models, techniques, efficiency and precision, etc.

More recent work focus on other sources of timing side-channels, induced by micro-architectural features (e.g., Spectre [32] and Meltdown [36]) or compilation (e.g., JIT-induced leaks [14]) where provably leakage-free programs (or with slight leakages) may become vulnerable when they are taken into account [18]. Our work is within this category.

Micro-architectural features allow new timing side-channel attacks such as Spectre, Meltdown and variants thereof [17, 18, 40, 45, 48]. This problem has been recently studied [18, 26–28, 30, 49, 51, 54–56], where speculative execution semantics, notions of constant-time under the new semantics, detection and mitigation approaches, etc, have been proposed. Among them, Blade 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 [10] and subsequently implemented in the verified compiler CompCert [8]. However, they disallow secret branches, increasing the difficulty of implementing constant-time programs. Follow-up work includes constant-resource preserving compilation [9]. However, neither of them considered JIT compilation which is far more complex than the static compilation.

The work on JIT-induced timing channel is very limited. The work close to ours is [13, 14, 16]. The JIT-induced leaks proposed in [14] demonstrated how JIT compilation can be leveraged to mount timing side-channel attacks. A fuzzing approach was proposed to detect JIT-induced leaks [16]. However, it can neither prove free of JIT-induced leaks nor mitigate JIT-induced leaks. The three strategies (i.e., NOJIT, DisableC2 and MExclude) proposed in [13] have been discussed and compared in Section 2.2 and Section 6.2.

8 Conclusion and Future Work

We have presented an operational semantics and a formal definition of constant-time programs under JIT compilation,

based on which we have proposed an automated approach to eliminate JIT-induced timing side-channel leaks. Our approach systematically detects potential leaks via a precise information flow analysis and eliminates potential leaks via a fine-grained JIT compilation. We have implemented our approach in the tool DeJITLeak. The evaluation shows that DeJITLeak is more effective than existing solutions and provides a trade-off between the security and performance. The lightweight variant DeJITLeak$_{\texttt{light}}$ of DeJITLeak pushes the limit of overhead further with comparable effectiveness.

In the future, we plan to improve our approach by taking into account other JIT optimizations and CPU-level optimizations. The efficiency could also be improved by refining the granularity of our fine-grained JIT compilation, e.g., methods invoked in both branches of a secret branch point could be inlined simultaneously which would not break the balance of the two branches.

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] 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

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

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

[14] 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

[15] 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

[16] 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

[17] 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.

[18] 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

[19] 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

[20] 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

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

[22] 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.

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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

[28] 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

[29] 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

[30] 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

[31] Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. 2011. A Trace-based Java JIT Compiler Retrofitted from a Method-based Compiler. In Proceedings of the 9th International Symposium on Code Generation and Optimization (CGO). 246–256.

[32] 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

[33] 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

[34] 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

[35] 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

[36] 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.

[37] 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

[38] 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

[39] 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

[40] 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

[41] 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

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

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

[44] 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

[45] 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

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

[47] 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

[48] 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

[49] 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

[50] 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

[51] 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

[52] 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

[53] 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

[54] 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

[55] 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

[56] 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

A Supplementary Material

This is supplementary material of the submission entitled "DeJITLeak: Eliminating Just-In-Time Compilation Induced Timing Side-Channel Leaks".

A.1 Demonstration of Branch Prediction (Tbran)

To demonstrate the Tbran timing side-channel, we use a simple method verifyPin shown in Figure 6a. It accepts a user-supplied parameter input and returns false if input!=pin and true otherwise. Clearly, it is not vulnerable without JIT compilation. However, verifyPin indeed is vulnerable to Tbran. To trigger Tbran, we execute verifyPin 50,000 times with pin=0xdeadbeaf, where input is randomly generated with probability of $\frac{1}{8}$ being 0xdeadbeaf. The execution of the if-branch is more often than that of the else-branch, triggering the branch predication optimization. After JIT compilation, the if-branch will take less time than the else-branch. This is justified by executing verifyPin 2,000 times (1,000 times for input==pin and 1,000 times for input!=pin), and computing the distribution of execution time, shown in Figure 6b. As a cross reference, Figure 6c shows the distribution of execution time with JIT compilation disabled. We can observe that the executions time of two branches is almost the same when the JIT compilation is disabled, while there exists a gap between the execution time of two branches when the JIT compilation is enabled. This difference allows the adversary to determine if input is the correct pin without knowing the return value of the method in a "blind" scenario.

By disabling the branch prediction of the conditional statement, the Tbran-induced leakage is prevented, justified by the execution time shown in Figure 6d. Note that the verifyPin method could be JIT compiled when our approach is applied.

A.2 Demonstration of Method Compilation (Tmeth)

To demonstrate the Tmeth timing side-channel, consider the checkSecret method shown in Figure 7a, which is extracted and simplified from the STAC canonical program *category1* [46]. It is not vulnerable in interpreted mode when the execution time of consume1 and consume2 is identical. However, if one of the methods is compiled or inlined at runtime while the another one is not, the branches will have unbalanced execution time, thereby introducing a new timing side-channel. To justify this, we invoke checkSecret only once with $n = 500$ and guess being a very small number. This enforces consume1 in the if-branch to be invoked 250,000 times, large enough to trigger the method compilation of consume1. After that, as shown in Figure 7b, for each input guess, if the execution time is small, we can deduce that guess≤secret, while if the execution time is large, we can deduce that guess>secret. As a cross reference, Figure 7c shows the distribution of execution time with JIT compilation disabled, indicating that the execution time of the branches is similar. This side-channel is very stable, as the adversary can repeatedly first invoke checkSecret in advance by passing a very small number which triggers the method compilation of consume1, then the subsequent invocation of checkSecret uses some data to check if it is correct, leading to a leak. To speed up the guess-and-check process, the adversary could use a binary search.

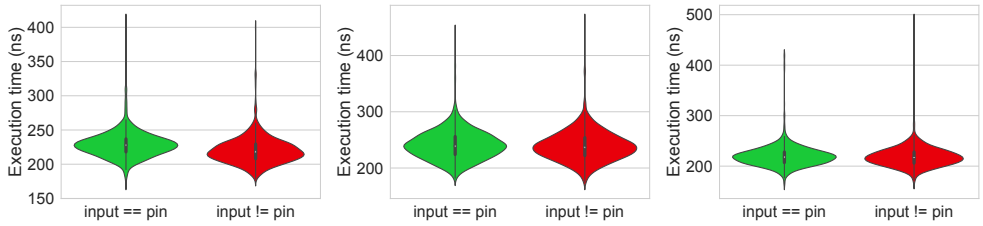
By disabling the JIT compilation of the consume1 and consume2 methods, the leak induced by Tmeth is prevented, justified by the execution time shown in Figure 7d. Note that the checkSecret method could be JIT compiled when our approach is applied.

A.3 Full Explanation of the Semantics

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.

Instruction return ends the execution of the current method, returns the top value v of the current operand stack, either by pushing it on top of the operand stack of the caller and re-executes the caller from the return site if the current method

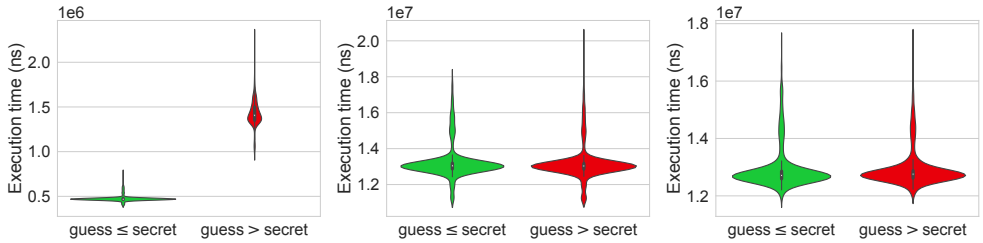
```
boolean verifyPin(int input){
    boolean equal= false;
    if (input != pin) {
        equal = false;
    } else {
        equal = true;
    }
    return r;}
```



(a) The verifyPin method **(b)** JIT enabled **(c)** JIT disabled **(d)** Mitigated

Figure 6. The verifyPin method and its execution time with JIT enabled and disabled under TBRAN

```
void checkSecret(int guess) {
  if (guess <= secret) {
    for (int i = 0; i < n; i++)
      for (int t = 0; t < n; t++)
        consume1();  }
  else {
    for (int i = 0; i < n; i++)
      for (int t = 0; t < n; t++)
        consume2();  }
}
```



(a) The checkSecret method **(b)** JIT enabled **(c)** JIT disabled **(d)** Mitigated

Figure 7. The checkSecret method and its execution time under TBRAN

is not the entry point, or enters a final configuration (h, v) if the current method is the entry point.

Instruction deopt md deoptimizes 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. We assume that 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 method invocation invoke m' depends on the directive $\mathbf{d}$. If $\mathbf{d}$ is $\mathbf{d}_\emptyset$ then the instructions of m' in the code heap ch remain the same. If $\mathbf{d}$ is valid, namely, the optimized version $\mathcal{V}_{m''}$ after applying $\mathbf{d}$ has larger version number than that of the current version $\mathcal{V}_{m'}$, the new optimized version $m'' = \mathbf{d}(m')$ is stored in the code heap ch. After that, it pops the top $|\mathsf{argv}(m')|$ values from the current operand stack, passes them to the formal arguments $\mathsf{argv}(m')$ of m', pushes the calling context on top of the call stack and starts to execute m' in the code heap.

```
0: load x0
1: get  pin
2: sub
3: ifeq 6
4: push 0
5: goto 8
6: push 1
7: goto 8
8: return
```

Figure 8. Before branch prediction optimization

```
0: load x0
1: get  pin
2: sub
3: ifeq 6
4: push 0
5: return
6: push 1
7: goto 5
8: goto 5
```

Figure 9. After branch prediction optimization

A.4 Formal Definition of the New Method m' After Branch Prediction

If the profiling data show that the program favors the else-branch instead of the if-branch, the branch prediction optimization transforms the method m into a new method m_1 as follows:

- $m_1[i]$ becomes ifeq j'' (resp. ifneq j'') if $m[i]$ is ifeq j (resp. ifneq j), where $j'' = |m| - j' + j - 1$ is the point of the first instruction of the if-branch B'_t in m_1;
- the if-branch B'_t in m_1 is the if-branch B_t, but is moved to the end of the method and appended with goto $j-1$, namely, $m_1[|m| - j' + j - 1, |m-2|] = m[j, j'-1]$ and $m_1[|m-1|] = \text{goto } j-1$;
- the else-branch B'_f in m_1 is the else-branch B_f, where the last instruction goto j' is removed, namely, $m_1[i+1, j-2] = m[i+1, j-2]$;

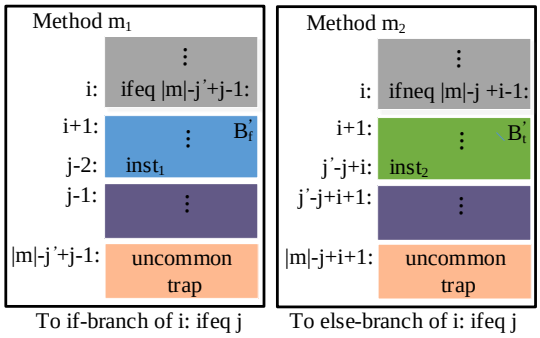


Figure 10. Optimistic compilation optimization

- furthermore, the target points of other conditional and unconditional jumps are revised accordingly.

Example A.1. Consider the method verifyPin shown in Figure 8, where the else-branch is at lines 4–5 and the if-branch is at lines 6–7. Note that the instruction at line 7 is added to balance the execution time of the two branches, which is done by manipulating the bytecode.

We can get T_{bp}(verifyPin, 3, else-b), as shown in Figure 9, where the else-branch is at line 4 and the if-branch is at lines 6–8. Obviously, the optimization unbalances the execution time of the two branches. Note that the instruction at line 8 is dead code, hence may be removed by other optimizations (e.g., peephole optimization), but the execution time of the two branches is still unbalanced.

A.5 Methods m_1 amd m_2 after Optimistic Compilation

Figure 10 (left-part) shows the method m' after the optimistic compilation optimization when the profiling data show that the if-branch almost never gets executed.

Figure 10 (right-part) shows the method m' after the optimistic compilation optimization when the profiling data show that the else-branch almost never gets executed.

A.6 Proof of Theorem 4.1

Theorem 4.1. Given a program P, if P is constant-time and $(\mathtt{prot}_1, \mathtt{prot}_2, \mathsf{sig}_P) \rhd P$, then P is JIT-constant-time under $(\mathtt{prot}_1, \mathtt{prot}_2)$-schedules.

Proof. Consider the program P. Assume P is constant-time and $(\mathtt{prot}_1, \mathtt{prot}_2, \mathsf{sig}_P) \rhd P$ holds. To prove that P is JIT-constant-time under $(\mathtt{prot}_1, \mathtt{prot}_2)$-schedules, we first introduce some notations.

Recall that for every method m and every valid directive $\mathbf{d} \in \mathbf{D}_m$, we denoted by $\mathbf{d}(m)$ the new optimized version of m after applying the directive $\mathbf{d}$. For a sequence of valid directives $\mathbf{d}_1 \cdots \mathbf{d}_n$ of m, we denote by $[\mathbf{d}_1 \cdots \mathbf{d}_n](m)$ the latest optimized version of m after sequentially applying the directives $\mathbf{d}_1 \cdots \mathbf{d}_n$, where $[\epsilon](m) = m$.

For each method m with $\mathsf{sig}_P(m) = (\mathsf{pt}_1, \mathsf{ht}_1, \mathsf{lt}_1) \hookrightarrow_m (\mathsf{ht}'_1, \tau)$, two configurations $c_0 = (\mathsf{ch}, \mathsf{h}, \langle 0, m, \rho, \epsilon\rangle, \epsilon)$ and $c'_0 = (\mathsf{ch}, \mathsf{h}', \langle 0, m, \rho', \epsilon\rangle, \epsilon)$ are $\mathsf{sig}_P(m)$-equivalent, denoted by $c_0 \simeq_{\mathsf{sig}_P(m)} c'_0$, if h and h' agree on the variables that have high security level H in ht_1, and ρ and ρ' agree on the variables that have high security H level in lt_1.

To prove that P is JIT-constant-time under $(\mathtt{prot}_1, \mathtt{prot}_2)$-schedules, we first prove the following lemma:

Lemma A.2. *Consider a method m and two configurations $c_0 = (ch, h, \langle 0, m, \rho, \epsilon\rangle, \epsilon)$ and $c'_0 = (ch, h', \langle 0, m, \rho', \epsilon\rangle, \epsilon)$ such that $c_0 \simeq_{sig_P(m)} c'_0$. Given a sequence of valid directives $\mathbf{d}_1 \cdots \mathbf{d}_n$ of m w.r.t. the policy $(\mathtt{prot}_1, \mathtt{prot}_2)$, let c_2 and c'_2 be two configurations such that $c_2 = (ch, h, \langle 0, [\mathbf{d}_1 \cdots \mathbf{d}_n](m), \rho, \epsilon\rangle, \epsilon)$ and $c'_2 = (ch, h', \langle 0, [\mathbf{d}_1 \cdots \mathbf{d}_n](m), \rho', \epsilon\rangle, \epsilon)$. We have:*
if $cf(c_0 \Downarrow_{\mathbf{d}^\star_\emptyset} c_1) = cf(c'_0 \Downarrow_{\mathbf{d}^\star_\emptyset} c'_1)$,
then $cf(c_2 \Downarrow_{\mathbf{d}^\star_\emptyset} c_1) = cf(c'_2 \Downarrow_{\mathbf{d}^\star_\emptyset} c'_1)$.

We prove this lemma by induction on the length of the sequence $\mathbf{d}_1 \cdots \mathbf{d}_n$. The base case follows from the fact that $[\epsilon](m) = m$. We consider the inductive step $n \geq 1$.

Let $c_3 = (\mathsf{ch}, \mathsf{h}, \langle 0, [\mathbf{d}_1 \cdots \mathbf{d}_{n-1}](m), \rho, \epsilon\rangle, \epsilon)$ and $c'_3 = (\mathsf{ch}, \mathsf{h}', \langle 0, [\mathbf{d}_1 \cdots \mathbf{d}_{n-1}](m), \rho', \epsilon\rangle, \epsilon)$. Then, $c_3 \simeq_{\mathsf{sig}_P(m)} c'_3$. By applying the induction hypothesis: we get that $\mathsf{cf}(c_3 \Downarrow_{\mathbf{d}^\star_\emptyset} c_1) = \mathsf{cf}(c'_3 \Downarrow_{\mathbf{d}^\star_\emptyset} c'_1)$. If $\mathbf{d}_n = \mathbf{d}_\emptyset$, the result immediately follows. We consider $\mathbf{d}_n = (t, \omega)$.

Let $c_4 = (\mathsf{ch}, \mathsf{h}, \langle 0, t([\mathbf{d}_1 \cdots \mathbf{d}_{n-1}](m)), \rho, \epsilon\rangle, \epsilon)$ and $c'_4 = (\mathsf{ch}, \mathsf{h}', \langle 0, t([\mathbf{d}_1 \cdots \mathbf{d}_{n-1}](m)), \rho', \epsilon\rangle, \epsilon)$. Then, $c_4 \simeq_{\mathsf{sig}_P(m)} c'_4$.

Since the directive $\mathbf{d}_n$ respects the $(\mathtt{prot}_1, \mathtt{prot}_2)$ policy and $(\mathtt{prot}_1, \mathtt{prot}_2, \mathsf{sig}_P) \rhd P$, all the methods specified in t can be invoked only with the path context L. This implies that the subsequences of executed instructions of the inlined methods are the same in the JIT-executions $c_4 \Downarrow_{\mathbf{d}^\star_\emptyset} c_1$ and $c'_4 \Downarrow_{\mathbf{d}^\star_\emptyset} c'_1$, we get that $\mathsf{cf}(c_4 \Downarrow_{\mathbf{d}^\star_\emptyset} c_1) = \mathsf{cf}(c'_4 \Downarrow_{\mathbf{d}^\star_\emptyset} c'_1)$. Note that we have already assumed that the cost equivalence of bytecode instructions are preserved in native code.

Let us now consider the sequence ω of optimizations to branch points. As $(\mathtt{prot}_1, \mathtt{prot}_2, \mathsf{sig}_P) \rhd P$, no branch points whose the path context or condition is H can appear in ω, thus all the branches of the branch points of ω are the same in $t([\mathbf{d}_1 \cdots \mathbf{d}_{n-1}](m))$ and $[\mathbf{d}_1 \cdots \mathbf{d}_n](m)$ (module the code format, i.e., bytecode vs. native code if $n = 1$ and $\mathsf{ch}(m)$ is bytecode). Since the subsequences of executed branches of the branch points with the path context L are the same in the JIT-executions $c_2 \Downarrow_{\mathbf{d}^\star_\emptyset} c_1$ and $c'_2 \Downarrow_{\mathbf{d}^\star_\emptyset} c'_1$, we get that $\mathsf{cf}(c_2 \Downarrow_{\mathbf{d}^\star_\emptyset} c_1) = \mathsf{cf}(c'_2 \Downarrow_{\mathbf{d}^\star_\emptyset} c'_1)$.

Now, we start to prove that P is JIT-constant-time under $(\mathtt{prot}_1, \mathtt{prot}_2)$-schedules.

Consider a pair of initial configurations (c_0, c'_0) of P with $c_0 \simeq_{\mathsf{pub}} c'_0$ and $c_0 \simeq_{\mathsf{ch}} c'_0$. Let c_1 and c'_1 be the configurations obtained from c_0 and c'_0 by replacing the code heap with the bytecode version. Then, there exists a valid $(\mathtt{prot}_1, \mathtt{prot}_2)$-schedule $\mathbf{d}^\star$ such that the code heap in c_0 and c'_0 is equal to

the bytecode version after applying $\mathbf{d}^\star$. Since P is constant-time, we get that $\mathsf{cf}(c_1 \Downarrow_{\mathbf{d}^\star_\emptyset} c) = \mathsf{cf}(c_1' \Downarrow_{\mathbf{d}^\star_\emptyset} c')$. By Lemma A.2, we have: $\mathsf{cf}(c_0 \Downarrow_{\mathbf{d}^\star_\emptyset} c) = \mathsf{cf}(c_0' \Downarrow_{\mathbf{d}^\star_\emptyset} c')$.

Consider a pair of valid $(\mathtt{prot}_1, \mathtt{prot}_2)$-schedules $\mathbf{d}_1^\star$ and $\mathbf{d}_2^\star$ for c_0 and c_0'. Then, only methods $m \in \mathbf{M} \setminus \mathtt{prot}_1$ can be JIT compiled or inlined, namely, only methods that are never invoked with the path context H can be JIT compiled or inlined. Since $(\mathtt{prot}_1, \mathtt{prot}_2, \mathsf{sig}_P) \rhd P$ and $c_0 \simeq_{\mathsf{pub}} c_0'$, we obtain that $\mathbf{d}_1^\star$ and $\mathbf{d}_2^\star$ must be the same.

We show that $\mathsf{cf}(c_0 \Downarrow_{\mathbf{d}_1^\star} c) = \mathsf{cf}(c_0' \Downarrow_{\mathbf{d}_1^\star} c')$ by induction on the number of non-$\mathbf{d}^\star_\emptyset$ directives in $\mathbf{d}_1^\star$. The base case follows from the fact that $\mathsf{cf}(c_0 \Downarrow_{\mathbf{d}^\star_\emptyset} c) = \mathsf{cf}(c_0' \Downarrow_{\mathbf{d}^\star_\emptyset} c')$. For the inductive step, we assume that $\mathbf{d}_3^\star$ is the $(\mathtt{prot}_1, \mathtt{prot}_2)$-schedule obtained from $\mathbf{d}_1^\star$ by replacing the last directive $\mathbf{d} \in \mathbf{D}_m$ of $\mathbf{d}_1^\star$ with the directive $\mathbf{d}_\emptyset$. By applying the induction hypothesis, we get that $\mathsf{cf}(c_0 \Downarrow_{\mathbf{d}_3^\star} c) = \mathsf{cf}(c_0' \Downarrow_{\mathbf{d}_3^\star} c')$. By Lemma A.2, we get that $\mathsf{cf}(c_0 \Downarrow_{\mathbf{d}_1^\star} c) = \mathsf{cf}(c_0' \Downarrow_{\mathbf{d}_1^\star} c')$. □