University of California
Santa Barbara

Static and Dynamic Side Channels in Software

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Tegan Siobhan Brennan

Committee in charge:

Professor Tevfik Bultan, Chair
Professor Subhash Suri
Professor Giovanni Vigna

September 2020

The Dissertation of Tegan Siobhan Brennan is approved.

Professor Subhash Suri

Professor Giovanni Vigna

Professor Tevfik Bultan, Committee Chair

August 2020

Static and Dynamic Side Channels in Software

by

Tegan Siobhan Brennan

76

In memory of my beloved grandmother, Iris May Marrs

Acknowledgements

I would like to express my deepest thanks to my Ph.D. advisor, Tevfik Bultan, whose encouragement and mentorship have been invaluable to me. Throughout my years in the Verification Lab at UCSB, Tevfik has been exceptionally supportive and generous with his advice. It has been a joy to learn about scholarship, leadership and pedagogy from his example.

I am thankful to my committee members, Subhash Suri and Giovanni Vigna, for providing me with feedback, support and encouragement as I reached milestones in my Ph.D.

My time researching at UCSB was deeply enriched by the amazing group of scientists I collaborated with. I would especially like to thank Nicolás Rosner, who I am immensely grateful to have had the opportunity to work with and from whom I've learnt so much about initiative and work ethic. I would also like to thank Matthew Cieslak for his scientific insight and excellent communication. The perspectives, company and support of the members of the Verification Lab — Lucas, Baki, Bo, Nico, Nestan, Will, Seemanta, and Burak — made my time in lab not only intellectually exciting but full of camaraderie as well.

I am deeply grateful to Miroslav Gavrilov for sharing his reflections, creativity and humor with me over our many late-night conversations and for the memories of our friendship I cherish. I would like to thank Zachary King for his unhesitating generosity, steadfast support and dependable partnership. I have grown so much from his inquisitive mind and compassionate presence.

I was able to find another space for growth with the Santa Barbara dance and aerial communities. Through them, I found my own passion, creativity and diligence. I am deeply grateful to my teachers and to those I share my practice with.

I would finally like to thank my wonderfully supportive friends and family, especially my amazing mother and my beloved granny. Your understanding, belief, kindness and humor have meant the world to me during this journey.

Curriculum Vitæ

Tegan Siobhan Brennan

Education

2020	Ph.D. in Computer Science (Expected), University of California, Santa Barbara.
2014	M.Sc. in Computational Science and Engineering, University of California, Santa Barbara.
2012	B.A. in Mathematics, Princeton University.

Publications

Tegan Brennan, Seemanta Saha, Tevfik Bultan. *JVM Fuzzing for JIT-Induced Side-Channel Detection.* To appear in Proceedings of the 42nd International Conference on Software Engineering, Seoul, South Korea (**ICSE 2020**).

Tegan Brennan, Nicolas Rosner, Tevfik Bultan. *JIT Leaks: Inducing Timing Side Channels Through Just-in-Time Compilation.* To appear in Proceedings of the 41st IEEE Symposium on Security and Privacy, San Francisco, California, USA (**S&P 2020**).

William Eiers, Seemanta Saha, **Tegan Brennan**, Tevfik Bultan. *Subformula Caching for Model Counting and Quantitative Program Analysis.* Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, San Diego, California, USA (**ASE 2019**).

Abdulbaki Aydin, William Eiers, Lucas Bang, **Tegan Brennan**, Miroslav Gavrilov, Tevfik Bultan, Fang Yu. *Parameterized Model Counting for String and Numeric Constraints.* Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista, FL, USA (**FSE 2018**) 400–410.

Tegan Brennan, Seemanta Saha, Tevfik Bultan, Corina S. Pasareanu. *Symbolic Path Cost Analysis for Side Channel Detection.* Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Amsterdam, The Netherlands (**ISSTA 2018**) 27–37.

Matthew Cieslak, Wendy Meiring, **Tegan Brennan**, Clint Greene, Lukas Volz, Jean Marie Vettel, Subhash Suri, Scott Grafton. *Compositional measures of diffusion anisotropy and asymmetry.* 15th IEEE International Symposium on Biomedical Imaging, Washington, DC, USA (**ISBI 2018**) 123–126.

Tegan Brennan, Seemanta Saha, Tevfik Bultan. *Symbolic Path Cost Analysis for Side-Channel Detection.* Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, Gothenburg, Sweden (**ICSE 2018 Companion Volume Poster Track**) 424–425.

Matthew Cieslak, **Tegan Brennan**, Wendy Meiring, Lukas J. Volz, Alex Asturias, Subhash Suri, Scott Grafton. *Analytic tractography: A closed-form solution for estimating local white matter connectivity with diffusion MRI.* **NeuroImage 2018** 169: 473–484.

Tegan Brennan, Nestan Tsiskaridze, Nicolas Rosner, Abdulbaki Aydin, Tevfik Bultan. *Constraint Normalization and Parameterized Caching for Quantitative Program Analysis.* Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany (**FSE 2017**) 535–546.

Tegan Brennan. *Path Cost Analysis for Side Channel Detection.* Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA (**ISSTA Doctoral Symposium 2017**) 416–419.

Charles Hagwood, Javier Bernal, Michael Halter, John Elliott, **Tegan Brennan**. *Testing Equality of Cell Populations Based on Shape and Geodesic Distance.* **IEEE Transactions on Medical Imaging 2013**, vol. 32, no. 12, pp. 2230–2237.

Abstract

Static and Dynamic Side Channels in Software

by

Tegan Siobhan Brennan

We interact with computer systems daily if not hourly, trusting them with our sensitive data. Computer scientists build these systems and, as we do, we introduce often unintended sources of information leakage into our creations. Some of these appear benign, but in this increasingly connected world, information leaks have consequences. Side channels in software are a nuanced yet potentially devastating mechanism for information leakage. In this thesis, I address key concerns about side channel vulnerabilities in software: how they arise, how much information they leak, and whether their detection and mitigation can be automated.

I develop techniques to detect, quantify and mitigate side-channel vulnerabilities. I draw on an array of program analysis and testing techniques, such as taint analysis, model-counting, CFG analysis, fuzzing, and quantitative information flow. Importantly, I consider that side channel vulnerabilities can arise both statically at the source-code level and dynamically at runtime. With this second observation, I pioneer research into a previously unknown type of software side-channel, JIT-induced timing channels. These side channels arise dynamically at runtime due to an imbalance in the input distribution of a program, transforming the problem of side-channel vulnerabilities in software into one that needs to consider the set of possible runtime states.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Cyber-attacks stealing confidential information are becoming increasingly frequent and devastating as modern software systems store and manipulate greater amounts of sensitive data. Leaking information about private user data, such as the financial and medical records of individuals, trade secrets of companies and military secrets of states can have drastic consequences. Confidentiality of such private data is critical for users of these systems. Many software development practices, such as the encryption of packages sent over a network, aim to protect the confidentiality of private data by ensuring that an observer is unable to learn anything meaningful about a program's secret input from its public output.

Under these protections, the software system's main communication channels, such as the content of the network packets it sends, or the output it writes to a public file, should not leak information about the private data. However, many software systems still contain serious security vulnerabilities. Through observing non-functional side effects of software systems, a class of information leaks, referred to as side-channels, can capture secret information. Potential side-channels include those in execution time, memory usage, size and timings of network packets that are sent, and power consumption.

Generally, side channels arise from a correlation between secret input and a side-channel

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

Figure 1.1: A password-checking function.

observable. As an example, consider the password checking function, given in Figure 1.1. Here a guess and a stored password are compared character by character. The function returns false as soon as there is a mismatch between the characters of stored password and that of the guess. This introduces a correlation between the execution time of this function and the length of the prefix shared by the stored password and the guess. This correlation results in a side channel in time allowing an observer with the ability to time the function to determine the length of the prefix shared by the password and guess.

Though side-channel vulnerabilities have been known for many years [1], they are commonly thought of as impractical despite a growing number of demonstrations of realistic side-channel attacks that result in critical security vulnerabilities [2–4]. Recently, Spectre and Meltdown, side-channel vulnerabilities affecting most modern microprocessors, including Intel CPUs, have made headlines, causing a surge of interest in side-channel exploits. The resulting interest; however, has been highly focused on side-channel vulnerabilities in hardware while side channels in software remain overlooked despite their potential to introduce devastating security risks.

My work addresses this gap in the current research. My thesis is ultimately concerned with understanding:

- How side channels arise in software
- How much and what kind of information they leak
- How observable of a channel they provide
- Whether the above can be determined automatically

One aspect of my work is concerned with how these correlations arise. In the first chapter of my thesis, I address how the source code of a program, such as the code given in Figure 1.1, can introduce these correlations. I develop a static, scalable technique for the detection of side channels in software and discuss its implementation in prototype tool, CoCo-Channel. CoCo-Channel decomposes the control flow graph of the program into nested branch and loop components, and compositionally assign a symbolic cost expression to each component. Queries to a satisfiability solver on the difference between possible cost values of a component allow us to detect the presence of imbalanced paths (with respect to observable cost) through the control flow graph. When combined with taint analysis that identifies secret-dependent conditional statements, my technique answers the following question: Does there exist a pair of paths in the program's control flow graph, differing only on branch conditions influenced by the secret, that differ in observable side-channel value by more than some given threshold? I end the chapter with a comparison of CoCo-Channel to other program analysis tools for side-channel detection and discuss their respective results of well-established datasets.

CoCo-Channel and all other current analysis tools for side-channel analysis either implicitly or explicitly consider the cost of a program path as fixed, modulo the inevitable system noise for observables such as execution time. The underlying assumption is that the source code is the only determiner of the side-channel observable. This is not the case. Just-in-time (JIT) compilation, which is crucial to the performance of modern programming languages such as Java and Javascript, can introduce timing side channels into deceptively secure-looking code fragments when it attempts to optimize paths it deems "hot". For example, the Java Virtual Machine tracks how often each branch of a conditional branch instruction is taken, and uses this profiling data when optimizing a method to generate native code favoring the more common branch. As a result, JIT compilation can introduce timing side channels in cases where the input distribution to the program is non-uniform.

In the next chapter of my thesis, I introduce and address this class of side-channel vulnerabil-

ities. I develop a series of vulnerability templates characterizing code vulnerable to JIT-induced side-channels and a series of attack models under which these side-channels can be introduced and leveraged. I analyze a series of built-in Java and Javascript methods to determine the susceptibility of code segments matching the vulnerability templates under different attack models. The experimental results show that both runtimes evaluated are suspectible to JIT-induced side channels. I also successfully induce JIT-based side-channels in the Apache Shiro security framework and the GraphHopper route planning server that are large enough in magnitude to be observable over the public internet.

These side-channels are not detectable with any current program analysis approach. They are introduced dynamically during runtime through a bias in the input distribution to the program. Thus, a program's behavior is the result not only of its source code but of its input history. In the next chapter of my thesis, I present a technique for automatically detecting such JIT-induced timing side channels in Java programs. I first introduce patterns to detect partitions of secret input potentially separable by side channels. Then I present an automated approach for exploring behaviors of the Java Virtual Machine (JVM) to identify states where timing channels separating these partitions arise. I evaluate my technique on three datasets used in recent work on side-channel detection, including those used by CoCo-Channel. I find that many code variants labeled "safe" with respect to side-channel vulnerabilities are in fact vulnerable to JIT-induced timing side channels. These results directly contradict the conclusions of four separate state-of-the-art program analysis tools for side-channel detection. Not only does this show that JIT-induced side channels can be detected automatically but also that any technique that does not consider the impact of the runtime on side-channel observables is insufficient in determining side-channel vulnerability.

These JIT-induced side channels have considerable potential to leak information and therefore pose a threat to the privacy of user data. At the same time, just-in-time compilation is crucial to the performance on many modern programming languages. In my next thesis chapter,

I explore the challenge of mitigating JIT-induced side channels. I aim at developing mitigation techniques for this class of side-channel vulnerability that still allow the runtime to leverage just-in-time compilation for high performance. I introduce three different strategies for mitigation and empirically evaluate their impact on both information leakage and performance on built-in functions from the Java standard library. Based on those results, I propose introduce additional, more refined mitigation strategies and argue for their effectiveness.

Side-channel detection tools more generally are concerned only with the existence of side-channel vulnerabilities, reporting if there is *any* information about secret input that may be leaked from the public side-channel observable. In practice, it is often acceptable and occasionally even necessary for some information to be leaked. In an election protocol, for example, the individual votes should be secret, but the tally of votes should be known by the public. So how much leakage is acceptable? Quantitative program analyses provide a means for understanding the capacity of side channels to leak sensitive information. One such analysis, quantitative information flow, uses information-theoretic measures to quantify how much information about a program's secret input can be expected to be learned from observation of a side-channel observable. Its enabling technology is model-counting constraint solvers. While this technical analysis is powerful, it is expensive! Model-counting queries are at least as expensive as their satisfiability counterparts. To help mitigate the expensive of these queries, I developed Cashew, a caching framework for model-counting queries. In my next thesis chapter, I will present Cashew along with an experimental evaluation of the improvement in model-counting constraint solving enabled by Cashew. Integral to Cashew's performance is the constraint normalization procedure I developed, under which the cardinality of the solution set of a constraint, but not necessarily the solutions set itself, is preserved.

1.0.1 Contributions

My thesis contributes the following:

- A scalable, static analysis technique to detect side channels introduced in software through an imbalance present in a program's source code.
- The motivation and systemtic exploration of a different class of side channels introduced dynamically at runtime through just-in-time compilation.
- An automated technique for inducing JIT-based side channels into systems.
- Contributions to model-counting constraint solvers that allow for scalable quantatative analyses to answer the question of how much information a side channel leaks.

In the end these contributions are aimed towards answering the considerations set forth in the prior section: (1) How side channels arise in software (2) How much and what kind of information they leak (3) How observable of a channel they provide and (4) Whether the above can be determined automatically.

Chapter 2

Symbolic Path Cost Analysis for Side-Channel Detection

2.1 Introduction

In this chapter, I address the automatic detection of side-channel vulnerabilities through analysis of a program's source code. Side channels arise in software systems when there is a detectable correlation between program paths and external observables, such as execution time or memory usage. The password checking function discussed in the Introduction (Figure 1.1) illustrates how these correlations can arise in the source code of a program. In that example, the difference in execution time across program runs is due to different program behavior at a branch condition in the source code. This observation that side channels arise due to varied behaviour at conditionals related to secret information motivates my technique.

Given a set of secret variables, a type of side-channel, and a program, our goal is to detect the set of branch conditions responsible for potential side-channels of the given type in the program, and generate a pair of witness paths in the control flow graph for the detected side-channel. My technique achieves this by analyzing the control flow graph of the program with respect to a

cost model (such as time or memory usage), and identifies if a change in the secret value can cause a detectable change in the observed cost of the program behavior. It also generates a pair of witness paths in the control flow graph, differing only on the branch conditions influenced by the secret, whose observable output under the given side-channel differs by more than some user defined threshold. I present an extension to this technique to determine the minimum number of loop iterations necessary for a side-channel to be visible in the program as well as the maximal cost difference between any two paths through the control flow graph. I implemented this approach in a prototype tool, CoCo-Channel (COmpositional COnstraint-based side-CHANNEL analyzer), for analyzing Java programs. My approach consists of following four phases:

(1) Annotated Control Flow Graph Extraction and Decomposition: We use taint analysis to identify the branch conditions in the program that are influenced by the secret information. Then, we extract a reducible control flow graph where each basic block is annotated with a cost based on the cost model used. We decompose the control flow graph to a set of nested loop and branch components to facilitate compositional symbolic cost expression construction.

(2) Symbolic Cost Expression Construction: We compositionally construct symbolic cost expressions for components based on their nesting relationships. We use one integer variable per branch and loop component, that encode which branch is taken and how many times a loop is executed, respectively.

(3) Constraint Solver Query Generation: We generate queries for constraint solvers that are satisfiable if only if there exist two paths in the control flow graph, that differ only on the secret-dependent branch conditions, whose observable output under the given side-channel differs by more than some user defined threshold. As an extension, we also generate optimization queries that identify two such paths while minimizing the number of loop executions or maximizing the cost difference.

(4) Feasibility Analysis: Not all paths through a control flow graph are executable. To

reduce the false positives this potentially introduces into our analysis, we perform two types of feasibility checks. The first determines whether the witness paths themselves are executable and the second generates constraints that must be satisfied in order for a pair of individually feasible witness paths to be executable on the same public input.

After a motivating example, I will explain phases of my analysis in order, followed with experimental evaluations and related work on side-channel detection.

2.2 A Motivating Example

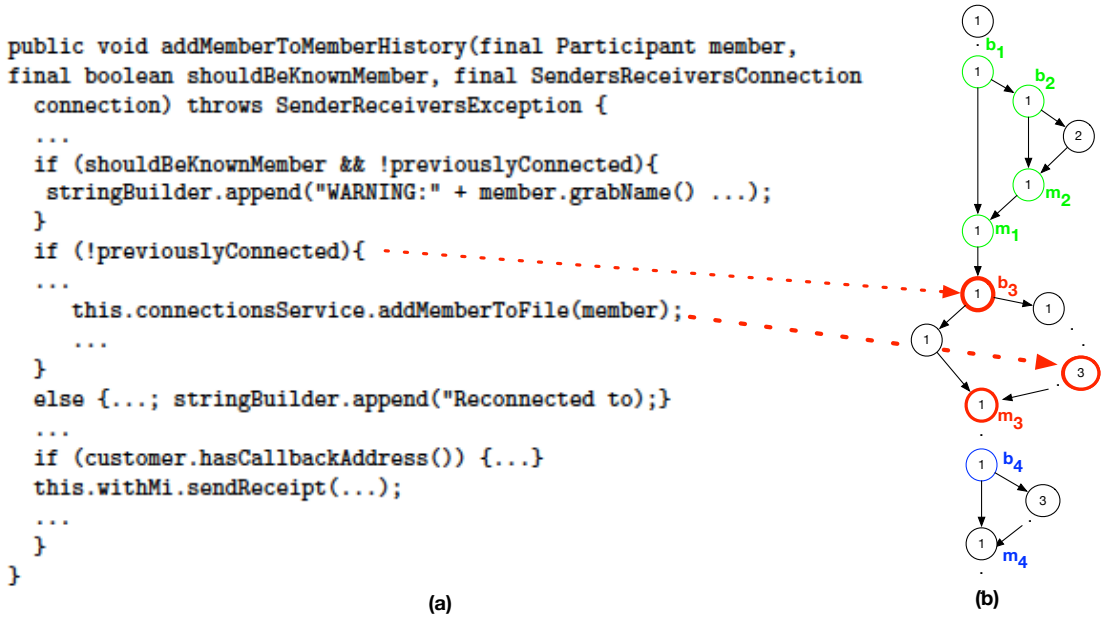


Figure 2.1: (a) Program vulnerable to a timing side-channel. (b) Control flow graph representation of the program. Arrows indicate the relationship between the code and resultant graph structures.

Consider the application code given in Figure 2.1(a) and its corresponding control flow graph in Figure 2.1(b). This code was extracted from a peer-to-peer messaging application provided as part of the DARPA STAC (Space-Time Analysis for Cybersecurity) Program [5,6] and contains

a side- hannel in time allowing an attacker to determine whether two users in the process of connecting have been previously connected. I demonstrate how an analysis of the program paths and their costs might detect the side-channel and the branch condition that introduces it.

Annotation. First, we identify the branch conditionals impacted by the secret value. There are three such conditionals denoted by b_1, b_2 and b_3 in the control flow graph. We ascribe to each node a cost. For simplicity of presentation, we use a simple cost model where a node can take one of three costs. 1 denotes an inexpensive node, 2 a moderately expensive node, and 3 an expensive node. In Figure 2.1, these costs are shown in the center of each node.

Decomposition. We decompose the annotated graph into a set of nested or disjoint components. In this example, all components are determined by the branch conditionals. The first of the four components present in the given control flow graph is determined by the branch node b_2 and its merge node m_2. This component is nested within the component determined by the branch node b_1 and its merge node m_1. The components determined by the pairs (b_3, m_3) and (b_4, m_4) constitute the remaining components of the graph.

Symbolic Cost Expression Construction. Our analysis compositionally derives a symbolic cost expression and corresponding set of constraints for each component. Evaluating this cost expression on satisfying instantiations of its symbolic variables generates a set of costs that span the set of all possible costs of paths through the component. In our example, the cost expression for the component associated with b_2 is given by $0 \cdot v_{b2} + 2 \cdot (1 - v_{b2}) + 1 + 1$, where v_{b2} is a symbolic variable constrained to 0 or 1 and denotes which branch was taken at b_2. The cost expression for the component associated with b_1, $0 \cdot v_{b1} + (1 - v_{b1}) \cdot (0 \cdot v_{b2} + 2 \cdot (1 - v_{b2}) + 1 + 1) + 1 + 1$, reuses the cost expression of its nested component. The cost expressions for the other components are formed analogously to that of b_2.

Constraint Solver Query Generation. Queries on the cost expressions and the associated constraints on their variables can be formulated to determine the existence of two paths through any component differing only on secret-dependent branches whose costs differ by more than

some threshold. Since both b_1 and b_2 are secret-dependent, if there are two pairs of satisfying assignments to v_{b2} and v_{b1} such that the difference in the evaluation of the cost expression, b_1, $0 \cdot v_{b1} + (1 - v_{b1}) \cdot (0 \cdot v_{b2} + 2 \cdot (1 - v_{b2}) + 1 + 1) + 1 + 1$ is greater than the threshold, then there are two paths through the component satisfying this criteria. However, 1 and 2 are not expensive costs, meaning such a satisfying assignment is not possible. The next component to analyze is that of b_3 whose associated cost expression $1 \cdot v_{b3} + (1+3) \cdot (1 - v_{b3}) + 1 + 1$ *does* have two satisfying assignments ($v_{b3} = 0, v_{b3} = 1$) to its secret-dependent symbolic variable v_{b3} such that the difference in its evaluation under those assignments is greater than our threshold. From this satisfying assignment, we learn that the choice made at branch conditional b_3 impacts the cost of two paths through the component in an observable manner. Analyzing the component associated with b_4, we observe that the two resultant branches also differ significantly in terms of cost. However, b_4 is a non secret-dependent conditional meaning that this cost imbalance does not introduce a side-channel to the application. This is modeled by a constraint that v_{b4} must match in any two assignments of variables that are compared for cost differences. In the general case, results across disjoint components would be combined to determine if multiple non-nested branch conditions introduce a side-channel.

I have demonstrated how queries on the set of possible costs of paths through the control flow graph can aid in the detection of side-channels. Explicit enumeration of all costs is usually not possible on programs of a meaningful size. Thus, I introduce a technique to compositionally generate a set of symbolic expressions and constraints characterizing all possible observable costs, and formulate queries over these expressions that can be answered using constraint solvers.

2.3 CFG Extraction and Decomposition

The control flow graph of a program provides a representation of all paths that might be traversed during its execution. We augment the control flow graph of a program by 1) a cost

model and 2) secret-dependent tainting.

Cost Model. Each run of a program returns not only its specified output, but also an observable value denoting the side-channel measurements of that execution of the program. For a given type of side-channel, we use a **cost model** to obtain static approximations of the observables that can be determined without running the system under test. For example, a cost model for timing side-channels might return the number of bytecode instructions along the control flow path while a cost model for side-channels in space might return the number of bytes written to a file. For our analysis, we assume that each node of the control flow graph has been annotated with the value of its associated basic block under the desired cost model.

Secret-Dependent Tainting. In programs that are of interest to our analysis, some of the input may be considered secret or private. These secret inputs may impact the results of evaluating certain conditional statements during execution. I refer to these conditional statements as secret-dependent and assume that all nodes of the control flow graph whose basic blocks contain such a conditional statement are marked appropriately.

Control Flow Graph Decomposition The decomposition of the control flow graph into components provides both scalability and interpretability benefits. On one hand, it allows us to modularize both our construction of cost expressions and the solving of resultant queries. On the other, it provides insight into the minimal amount of difference necessary between two control flow paths to introduce a side-channel. We assume a control flow graph in which appropriate node splitting ensures that no node is both a branch and merge node and any entry to a loop has only one incoming edge. We define our components using dominator and post-dominator relationships [7, 8].

Loop Components. Each back edge of the control flow graph introduces a loop component. For a given back edge, (a, b), its associated loop component is the union of b and the set of

nodes that can reach a without going through b.

Branch Components. A branch node b is any node that has more than one outgoing edge. The merge node m of a branch node b is the immediate post-dominator of b. The branch component associated with b is union of m and the set of all nodes that can be reached from b without going through m. If b is nested within a loop component that does not include its merge node, we define its loop-local branch component according to its immediate post-dominator of the loop component. These loop-local branch components are considered as branch components henceforth.

Hierarchical Structure Using previously computed cost expressions in the construction of new ones is possible if our components have a clearly defined nesting structure, meaning that for any two components, exactly one of the following three conditions hold: 1) The components are disjoint. 2) One component is a strict subset of the other i.e. they are nested. 3) The components are identical.

For two loop components, this assumption holds for *reducible* control flow graphs [9]. To gaurantee nesting for branch components, we assume a sufficient additional assumption that all non-branch, non-merge nodes of a branch component have only immediate ancestors within that branch component. Semantically, this means that every entry into the body of an if statement must be through its conditional guard.

2.4 Symbolic Cost Expressions

We construct a symbolic cost expression and associated constraints for each component G of the control flow graph such that: 1) for each cost c obtained along some path through G there exists a corresponding satisfying assignment of symbolic variables such that the symbolic cost expression evaluates to c, 2) for each value c generated by a satisfying assignment of the symbolic variables in the cost expression, there exists a corresponding path through G with cost

c.

The function *Annotate*, denoted as $A(G) \to (E, F)$, takes a graph G as input and returns a symbolic cost expression E along with a constraint F on the symbolic variables in E. The addition of two tuples (E, F) and (E', F') and the multiplication of a tuple (E, F) by a symbolic expression over integer variables v are performed by the $\oplus$ and $\otimes$ operators respectively, and defined:

$$(E, F) \oplus (E', F') = (E + E', F \wedge F')$$

$$v \otimes (E, F) = (v \times E, F)$$

$A(G)$ is defined in terms of two update functions, *UpdateBranch* $U_b(G, v) \to (E, F)$ and *UpdateLoop* $U_l(G, v) \to (E, F)$, where v is a symbolic expression over integer variables.

Annotate, *UpdateBranch* and *UpdateLoop* are defined by the rules given in Figure 2.2. There are four possible types of graphs these functions can take as arguments. If G is a single node, we denote it with G_n and denote its cost as c_n. If G is the sequential composition of other components or single nodes, we call it G_s and write $G_s = G_1, G_2, \ldots$ where each G_i is either a component or a single node. If G is a branch component, we use G_b. For any G_b, G'_{b1} and G'_{b2} denote the two disjoint subgraphs formed when the branch node b and merge node m are removed from G_b. If G is a loop component, we use G_l, and G'_l is the subgraph formed when the back edge of G is removed. G refers to a graph that may be any of the above.

Intuitively, E and F are constructed by introducing a symbolic integer variable and its appropriate constraints for each branch and loop, which encodes the branch taken or number of loop iterations, respectively. This can be observed in the $A(G_b)$ and $A(G_l)$ rules (rules (3) and (4)) where a fresh symbolic integer variable v is created and used in the appropriate *Update* function.

Symbolic cost expressions and their associated constraints are constructed hierarchically. Applying A to some component G assumes the results of applying A to all nested components

$$
\begin{aligned}
A(G_n) &= (c_n, \top) && (2.1)\\
A(G_s) &= A(G_1) \oplus A(G_2) \oplus \ldots && (2.2)\\
A(G_b) &= U_b(G'_{b1}, v) \oplus U_b(G'_{b2}, 1-v) \oplus A(b) \oplus A(m) && (2.3)\\
&\quad \oplus (0, v = 0 \vee v = 1)\\
A(G_l) &= U_l(G'_l, v) \oplus (0, v \geq 0) && (2.4)\\
U_b(G, v) &= v \otimes A(G) && (2.5)\\
U_l(G_n, v) &= (v \times c_n, \top) && (2.6)\\
U_l(G_s, v) &= U_l(G_1, v) \oplus U_l(G_2, v) \oplus \ldots && (2.7)\\
U_l(G_b, v) &= U_l(G'_{b1}, h) \oplus U_l(G'_{b2}, v-h) \oplus (0, h \leq v) \oplus && (2.8)\\
&\quad U_l(b, v) \oplus U_l(m, v)\\
U_l(G_l, v) &= \oplus_{i=1}^{v} v_i \otimes A(G'_l) \oplus (0, ((v-i) > 0 \wedge v_i = 1) && (2.9)\\
&\quad \vee ((v-i) \leq 0 \wedge v_i = 0))
\end{aligned}
$$

Figure 2.2: Rules defining the *Annotate* (A), *UpdateBranch* (U_b), and *UpdateLoop* (U_l) functions.

of G. Below, we give examples demonstrating symbolic expressions constructed according to the rules shown in Figure 2.2.

Figure 2.3(a) shows the symbolic cost expression E constructed for a simple branch component. The integer variable v_b corresponds to the integer variable created for the branch node in rule (3) in Figure 2.2. v_b is constrained to 1 or 0 and intuitively denotes whether the if or the else branch is taken at node b, respectively.

Figure 2.3(b) shows the construction of the symbolic cost expression for a branch component with a nested branch component. The value $A(G_b)$ for the nested component G_b was computed in Figure 2.3(a) and is reused in the construction of $A(G_{b*})$ by another application of rule (3) in Figure 2.2.

Figure 2.3(c) shows the symbolic cost expression construction for a loop component with a nested branch component. $A(G_l)$ is constructed by first applying rule (7) in Figure 2.2) and then by applications of rules (6) and (8).

Figure 2.4(a) shows the symbolic cost expression E constructed for a simple loop component.

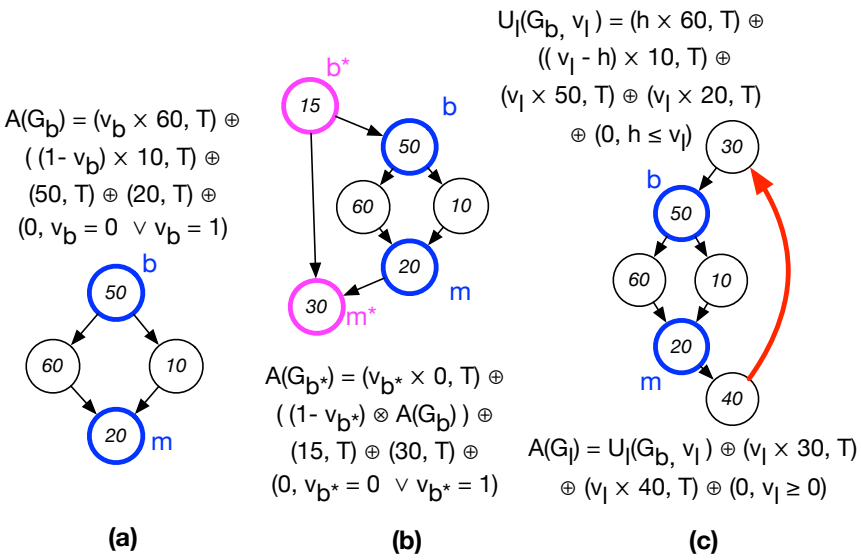


Figure 2.3: Symbolic cost expressions for (a) a simple branch component, and a branch nested within (b) another branch and (c) a loop.

The symbolic integer variable v_l, associated with the loop component (rule (4) in Figure 2.2), is constrained to be greater than or equal to 0 and denotes the number of times that the back edge of the loop is taken.

Figure 2.4(b) shows the symbolic cost expression construction for a loop component with a nested loop component. Here $A(G_l)$ is derived from applications of rules (7), (6) and (9).

Claim: Given any component G with $A(G) = (E, F)$, there is a satisfying assignment to the variables in F such that E evaluates to cost c if and only if there is a path through G with cost c.

Proof: We first show that for the cost c of any path through G, there is a satisfying assignment to the variables of F such that E evaluates to c. We do this through the construction of such an assignment. The cost of any path p through G is uniquely determined by the sequence of edges chosen when more than one outgoing edge could have been taken (i.e. at branch nodes). Following p through G, each time a branch condition is encountered, the value of the appropriate symbolic variable is set as follows.

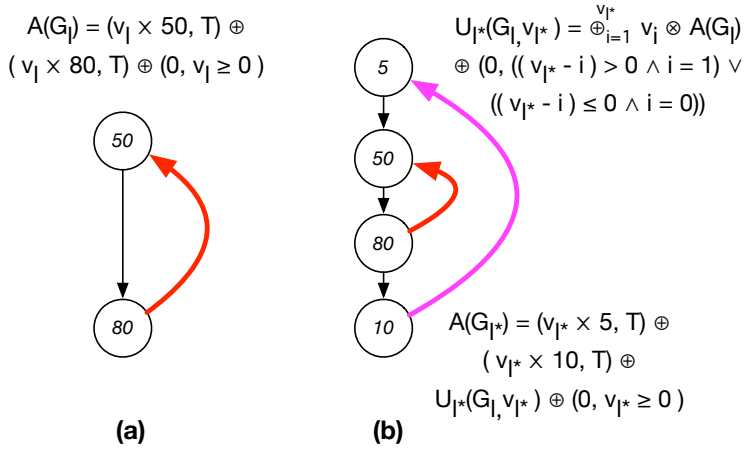


Figure 2.4: Symbolic cost expressions for (a) a simple loop component, and (b) a nested loop component within a loop component.

If the branch condition is the exit of a loop, then only if the exit is taken is the symbolic variable denoting the appropriate exit of the loop (if multiple exits are present) set to 1 and the value of the symbolic variable associated with the back edge of the loop set to the number of traversals of that edge. If the branch condition is nested within a loop, then a running tally is kept of how many times each branch is taken and the variable is set appropriately when the loop is exited. If the branch condition is not nested within a loop, then the update to its symbolic variable is simply the choice made in p. When loops are nested within other loops, the above annotation is performed for each iteration of the outer loop on the symbolic variables corresponding to that iteration. Symbolic variables assigned through this process will both satisfy F and evaluate in E to c.

Next, we show that any cost value obtained by evaluating E on a satisfying assignment of F is the cost of some path G. Let c be the cost generated by the E. By definition, there is some instantiation of the symbolic variables in F resulting in cost c. Let p be the path through component G dictated by that instantiation. In other words, each time a branch condition b is encountered along p, it is chosen according to the instantiation of the symbolic variable corresponding to b. If b is an exit for a loop, then it is taken if and only if the back edge of the

loop has been traversed the exact number of times specified by its associated symbolic variable and if b is the exit dictated by the instantiation of the symbolic variables corresponding to the possible loop exits. The cost of the path through G constructed in this manner will be c. ■

2.5 Query Generation

The construction of symbolic cost expressions allows for pertinent queries to be formulated concerning the set of costs they generate. To formulate these queries, we divide the set of symbolic variables in symbolic cost expressions to two types: secret-dependent and secret-independent. Recall that our control flow graph is annotated with information specifying which branch nodes are secret-dependent. We use this information to mark any symbolic variable associated with a branch component whose branch node is secret-dependent as secret-dependent, and any symbolic variable associated with the back edge of a loop with a secret-dependent exit node as secret-dependent. All other variables are marked as secret-independent. Given this, we generate two types of queries for side-channel analysis.

2.5.1 Existential Query

The first type of query we generate is aimed at answering the question: Do there exist two paths through the component differing only on conditionals impacted by the secret such that the difference between the two paths is greater than some threshold?

Given a component G, let (E, F) be the symbolic cost expression and the set of constraints generated by $A(G)$. The set of variables in F can be partitioned into two categories, the set of secret-dependent variables $\vec{v_s}$ and the set of secret-independent variables $\vec{v}$. We create two instances of (E, F): (E_1, F_1) and (E_2, F_2). The set of secret-dependent variables in F_1 is $\vec{v}_{s1}$ and that in F_2 is $\vec{v_{s2}}$. The set of secret independent variables $\vec{v}$ is shared across F_1 and F_2. We

then formulate and send the following query to an SMT-Solver:

$$\exists \vec{v_{s1}}, \vec{v_{s2}}, \vec{v} \text{ such that } |E_1 - E_2| > \delta \wedge F_1 \wedge F_2 \tag{2.10}$$

The satisfiability of this query illustrates the existence of two paths through G differing only on secret-dependent conditionals whose costs differ by more than the threshold δ. The satisfying assignments to variables $\vec{v}$, $\vec{v_{s1}}$ and $\vec{v_{s2}}$, denoted as $\vec{a}$, $\vec{a_{s1}}$ and $\vec{a_{s2}}$, provide a specification of such paths through G. This specification can be examined for its feasibility in the program and the process can be iterated to find new assignments if it is suspected to be infeasible.

Since the symbolic cost expressions constructed for multiple nested loops contain a sum over a symbolic variable (see rule (9) in Figure 2.2), a maximum value for this symbolic variable must be provided to ensure that our expression is within the theories supported by SMT Solvers. This value bounds the maximum number of loop iterations. As such, the unsatisfiability of our query only demonstrates the non-existence of two paths up to the given loop bound. Nevertheless, the analysis can be repeated across multiple bounds to increase certainty in the results.

Note that symbolic cost expressions may contain non-linear arithmetic expressions in the presence of nested loops, making the generated query potentially unsolvable by an SMT-Solver. Additionally, the number of generated variables grows exponentially with the nesting depth, leading to scalability concerns for components with highly nested loops. To address both of these challenges, we use a compositional approach to query generation that simplifies the cost expressions for nested components.

Compositional Query Solving. The symbolic cost expression generation technique we discussed earlier is compositional, and we exploit this to implement a compositional query solving technique that expedites the query solving process. For each component G nested within a secret-dependent component, we calculate the maximum and minimum cost paths through G

and, when applicable, two paths through G that differ only on secret-dependent conditionals and for which the cost difference is maximized. We call the assignment yielding the maximum cost value $\vec{a}_{max}$, that yielding the minimum $\vec{a}_{min}$, and that yielding the maximum difference $\vec{a}_{diff}$. The following claim allows us to determine the satisfiability of the existential query more cheaply using these values.

Claim: For any component, G, $A(G) = (E, F)$ for which $\exists \vec{v_{s1}}, \vec{v_{s2}}, \vec{v}$

such that $|E_1 - E_2| > \delta \wedge F_1 \wedge F_2$, there exist assignments from $\vec{a}_{max}$, $\vec{a}_{min}$, or $\vec{a}_{diff}$ to the variables corresponding to the nested components of G that also satisfy $|E_1 - E_2| > \delta \wedge F_1 \wedge F_2$.

Proof: Let $\vec{a}_1$ and $\vec{a}_2$ be the two assignments such that $|E_1 - E_2| > \delta$ and assume, without loss of generality, that $\vec{a}_1$ generates the larger cost value. There are two possibilities: $\vec{a}_1$ and $\vec{a}_2$ either differ in the symbolic variable associated with G, or they do not. First, let us assume that they do. Let $\vec{v_1}$ be the set of variables excluding the variable associated with G, whose assignment contributes to the value of E_1, meaning that a change in their assignment could result in a difference in the value of E_1. Replace the assignment of every variable in $\vec{v_1}$ with its assignment in $\vec{a}_{max}$ and replace the assignment of every variable in the complement of $\vec{v_1}$ with its assignment in $\vec{a}_{min}$. The two resulting assignments differ in cost by at least as much as the original $\vec{a}_1$ and $\vec{a}_2$.

Now, let us assume that $\vec{a}_1$ and $\vec{a}_2$ do not differ in the variable associated with G. Replacing the assignment of every variable associated with a nested component of G by that of $\vec{a}_{diff}$ (ensuring that the more expensive of the two assignments in each component always contributes to the value of E_1) again results in a pair of assignments that differ by at least as much as the original. ■

We can similarly show that finding $\vec{a}_{max}$, $\vec{a}_{min}$, and $\vec{a}_{diff}$ for some component G can also be done using the $\vec{a}_{max}$, $\vec{a}_{min}$, and $\vec{a}_{diff}$ assignments of its nested components. The paths built in this manner; however, tend to be the most extreme and thus are potentially more likely to be infeasible in the program.

2.5.2 Optimization Query

The existential query formulated in Section 2.5.1 is aimed at addressing the boolean question of the existence of a side-channel observable at some threshold and for some number of loop iterations. As choosing a good value for either parameter is challenging a priori, we reformulate the existential query as an optimization problem to obtain additional insight into both the feasibility and observability of the side-channel.

Our first reformulation is the following: What is the minimum number of loop iterations needed for the difference in cost between two paths in the component differing only on secret-dependent conditionals to be greater than some threshold? Let G, E_1, E_2, F_1, F_2, $\vec{v}$, $\vec{v_{s1}}$ and $\vec{v_{s2}}$ be defined as in Eq. 2.10. We define $\vec{v_k}$ to be the set of all variables associated with a back edge in the union of $\vec{v}$, $\vec{v_{s1}}$ and $\vec{v_{s2}}$. We generate the following minimization query:

$$\textbf{minimize } \vec{v_k} \text{ such that } |E_1 - E_2| > \delta \wedge F_1 \wedge F_2 \tag{2.11}$$

The expression to optimize can easily be adapted to weigh the minimization of certain loops differently. While a maximum number of loop iterations must still be specified to ensure the appropriate translation to the SMT-Solver in the case of iteratively nested loops, this optimization query relaxes the importance of the loop bound. Additionally, the result of the minimization query can provide some insight into the realizability of the side-channel. Though not always the case, it is likely that a side-channel achievable with relatively few loop iterations is more likely to be realizable than one that requires a large number of loop iterations.

The second reformulation is the following: What is the maximum difference between the costs of two paths through the component differing only on secret-dependent conditionals? Again, let G, E_1, E_2, F_1, F_2, $\vec{v}$, $\vec{v_{s1}}$ and $\vec{v_{s2}}$ be defined as in Eq. 2.10 and 2.11. We generate the

following maximization query:

$$\textbf{maximize } |E_1 - E_2| \text{ such that } F_1 \wedge F_2 \tag{2.12}$$

The result of this query provides a greater understanding of the observability of the side-channel in cases where an analyst is less sure of an appropriate value for the threshold δ.

2.6 Feasibility Analysis

Assignments returned by SMT solvers to queries generated from symbolic cost expressions may be un-exploitable in the program. To mitigate the impact of false positives, we augment CoCo-Channel with two types of symbolic-execution based feasibility analyses.

2.6.1 Single Path Feasibility Analysis

We perform a guided symbolic execution along each witness path to check its individual feasibility. Here we constrain our search to the largest branch component containing the vulnerable component. There is exactly one path to the vulnerable component in this subgraph, so this expansion does not increase the size of the analysis. Since the symbolic execution is guided along a single path, the exponential path explosion commonly debilitating to symbolic execution is avoided. However, because we are assuming that the expanded component can be reached by any input, we may incorrectly determine that a path that is feasible in the expanded component but infeasible in the larger program is executable. Nevertheless, this focused feasibility test can still eliminate false positives that are due to infeasible paths.

In the case where CoCo-Channel specifies not just one, but a class of witness paths, we check the feasibility of one example path. If it is infeasible, another path from the class can be selected. While this process itself may be exponential, it can be terminated at any time

and an additional constraint may be added to CoCo-Channel's queries to disallow that class of paths from consideration as a witness path. If CoCo-Channel can find another pair of assignments demonstrating the side-channel, then it is possible that proving the feasibility of the side-channel over these witness paths may be simpler.

2.6.2 Relational Feasibility Analysis

Two paths might be individually feasible but not executable for the same public input. In the function **sanity_safe** (Figure 2.5), both taking the if branch and executing the while loop 1 time and taking the else branch and executing the while loop 5 times are feasible paths. However, they are not both feasible for the same public input (the value of b). The reason for this lies in the correlation over public input in the number of iterations of the two loops.

Let v_1 and v_2 be the symbolic variables associated with two distinct conditionals and V_1 and V_2 be the set of feasible values for v_1 and v_2 respectively. For two value assignments $a_1 \in V_1$, $a_2 \in V_2$, we use $\mathcal{P}(a_1, a_2)$ to denote there is no public input on which both a_1 and a_2 are executable.

The two conditionals are said to be correlated over public input if $\exists a_1 \in V_1, \exists a_2 \in V_2$ such that $\mathcal{P}(a_1, a_2)$. Given a set of k-paths, we call the problem of determining the correlated set of conditional statements over public input *k-path relational analysis*. Because CoCo-Channel returns a witness pair of paths, we are interested in particular in the 2-path relational analysis problem. Additionally, we constrain the broad problem of determining any correlation between conditionals to determining only their equivalence or non-equivalence. In other words to determine if $\forall a_1 \in V_1, \forall a_2 \in V_2, a_1 \neq a_2, \mathcal{P}(a_1, a_2)$ or if $\forall a_1 \in V_1, \forall a_2 \in V_2, a_1 = a_2, \mathcal{P}(v_1, v_2)$.

Assume two assignments $\vec{a}_1$ and $\vec{a}_2$. If $\vec{v}^*$ are the secret-dependent conditionals on which the two paths differ, then the only public-dependent conditionals we need consider lie in the components of $v_i \in \vec{v}^*$. Since we are concerned with the cost differential between the paths, we

```
public static boolean sanity_safe(int a, int b){
    int i = b; int j = b;
    if (b < 0) {return false;}
    if (a > 0) {while (i>0) {i--;} }
    else {while (j>0) {j--;}}
    return false;
}
```

Figure 2.5: **sanity_safe** example from BLAZER dataset.

additionally constrain our analysis to only those public-dependent conditionals whose value in $\vec{a}_1$ or $\vec{a}_2$ introduce a meaningful cost differential between the paths.

Let v_1 and v_2 be symbolic variables corresponding to such public-dependent conditionals whose values in $\vec{a}_1$ and $\vec{a}_2$ respectively introduce a cost differential in the component of $v_i \in \vec{v}^*$. Without loss of generality, we assume that $v_i = 1$ in $\vec{a}_1$ and $v_i = 0$ in $\vec{a}_2$. Note that v_1 and v_2 may be either branch or loop variables. We use symbolic execution to determine if there is a correlation between the conditionals associated with v_1 and v_2. First we instrument the bytecode of a program by introducing integer variables for v_i, v_1 and v_2 that increment each time the corresponding conditional is taken, resulting in either a loop counter or a branch indicator. We then duplicate the program under test and execute both versions of the program symbolically using the same symbolic public input and fresh symbolic private input for the second. Letting the suffix denote the version of the program, we then confirm whether any of the following properties can be violated:

$$v_{i1} = 1 \wedge v_{i2} = 0 \wedge v_1 = v_2$$

$$v_{i1} = 1 \wedge v_{i2} = 0 \wedge v_1 \neq v_2$$

If the first one is never violated, then v_1 and v_2 are correlated and equivalent. If the second one is never violated, then v_1 and v_2 are correlated and non-equivalent. In either case, we can add an additional constraint to our query reflecting this information.

2.7 Implementation and Experiments

We implemented our analysis in our tool CoCo-Channel [10], a Python prototype that performs the decomposition and annotation described in Sections 2.3 and 2.4 and interfaces with tools for control flow graph extraction and satisfiability and optimization querying.

Control Flow Graph Extraction. The extraction of the control flow graph was done using Janalyzer [11], an in-development research tool for the static analysis of Java Byte-code developed by Balasubramanian et. al at Vanderbilt.

Taint Analysis. Janalyzer was also used to perform the taint analysis needed by our approach. Static taint analysis is an area of ongoing research [12], and obtaining sound results without over approximation is challenging. We observed that for a sizable class of side-channel vulnerabilities, taint analysis based only on data dependency would suffice to correctly mark secret-dependent branches and would achieve a lower false positive rate than taint analysis based on both data and control dependency. While this introduces a source of unsoundness, we constrain our taint analysis to track only data-dependencies for our experiments.

Satisfiability and Optimization Queries. We integrated CoCo-Channel with the SMT-Solver Z3 [13] through Z3Py, the Z3 API in Python [14]. All queries of the type formulated in Section 2.5 are sent to Z3 in order to determine either their unsatisfiability or to obtain a satisfying or optimizing assignment.

Feasibility Analysis. We guided the search of the symbolic execution engine Symbolic Path-Finder [15] (SPF) along the pair of witness paths returned by CoCo-Channel to determine their individual feasibility. To determine their relational feasibility, we used javassist [16] to instrument the bytecode with the additional integer variables discussed in Section 2.6.2. We perform symbolic execution of the instrumented bytecode and find any violations of the properties given in Section 2.6.2 using SPF.

2.7.1 Experiments on the STAC Benchmark

We evaluate CoCo-Channel on a benchmark suite of fourteen examples from the DARPA Space/Time Analysis for Cybersecurity (STAC) program [5, 6]. All fourteen questions require determining the vulnerability of large Java client-server or peer-to-peer applications to timing side-channels. Each specifies a secret value, and we evaluate CoCo-Channel's ability to determine if there is a timing side-channel that leaks information about that secret.

Information about these applications is provided in Table 2.1. The STAC applications are non-trivial applications including web-forums, image-sharing applications, peer-to-peer chat programs, and peer-to-peer applications for hosting auctions. Multiple versions of the same application are provided in the STAC benchmark, which often differ in terms of their vulnerability to side-channels. The number following the application name refers to the version.

For each question, we manually marked the input or program variable corresponding to the specified secret and an entry point into the application. We then used Janalyzer to perform taint analysis from our the variable and extract the control flow graph from the entry point. Function calls were handled through inlining. Presently, CoCo-Channel does not handle recursion, but instead unrolls the recursive calls through iterative inlining until some specified depth is reached. For our experiments, we used a depth of 1. Our cost model was the number of bytecode instructions in a basic block. Since not all bytecode instructions take the same amount of time, we augmented our cost model through a manually compiled list of bytecode operations known to be more expensive. Our list consisted of write-to-file operations, the sleep instruction, and the expensive arithmetic operations (multiply, mod, div, exp, pow, sqrt) and their BigInteger counterparts. The cost of these operations was increased according to the values provided in [17] except for sleep which was increased by its argument.

CoCo-Channel was run on the resultant annotated control flow graph for a threshold of 1000 and a maximum loop depth of 20. All experiments were run on a single computer with an

Table 2.1: Application Statistics

Application	#Classes	#Methods	#Bytes_of_instruction
gabfeed_1	111	332	12180
gabfeed_2	69	283	8862
gabfeed_3	112	332	12143
snapbuddy_2	338	2224	81636
snapbuddy_3	1083	3823	120351
powerbroker_1	272	3158	61288
powerbroker_4	261	3146	59948
withmi_4	218	2527	46011
withmi_5	213	2526	45996
collab	26	121	7236
lawdb	43	173	9987

Intel Core i5-6600K CPU @ 3.50GHz and 32GB of RAM.

Results. The results for the STAC benchmark are given in Table 2.2. The **Name** column identifies the example under test. **Size** is the number of basic blocks in our extracted control flow graph. **Vulnerable** is DARPA's official verdict and **Correct** denotes if our results agree with DARPA's. **UC-CoCo Time** is the time taken by CoCo-Channel to verify the application without compositional solving and **CoCo Time** is the time taken when compositional solving is used. It is extremely clear from the timing results on the larger STAC programs that compositional solving is key to scaling our technique to realistically sized applications. In five out of the fourteen cases, the naive solving approach times out after thirty minutes while the compositional solving approach is able to find a pair of witness paths in less than 3 seconds. CoCo-Channel's analysis agrees with DARPA's official verdict in all but two cases. Importantly, no vulnerable case is incorrectly classified as non-vulnerable. Moreover, CoCo-Channel is able to correctly specify the component of the control flow graph that introduces the side-channel in each correctly identified vulnerable case.

Discrepancies. DARPA considers **gb2_mdp** non-vulnerable, but CoCo-Channel's detection of a potential side-channel caused us to further examine the problem. The interesting method is a modular exponentiation function using the Montgomery Powering Ladder [18] to

perform exponentiation. While a single iteration of the loop in this method is in fact non-vulnerable, a difference in a given bit of the secret exponent value does impact how certain variables are updated, which can lead to subsequent loop iterations taking a differing amount of time. A differential side-channel attack may be able to exploit this and the original presentation of the Montgomery Powering Ladder [18] acknowledges its potential vulnerability to such side-channel attacks. Thus, we are not convinced that there is no side-channel in **gb2_mdp**, but perhaps is not strong enough to recover the entire key or not exploitable within the application.

The other discrepancy concerns **withmi5_connect**. Manual inspection of the code reveals that the the branch conditional marked vulnerable by CoCo-Channel is both secret-dependent and that the differences in work between the two resultant branches is significant (including a file write). However, we determined that whether or not the more expensive path is taken is not visible based on the timing of network traffic because there are no network packets sent after the choice on conditionals is made, making the side-channel unexploitable. Such a distinction is out of the scope of our analysis at present but motivates a potential extension of CoCo-Channel to include markers denoting where side-channel information is observable (such as when packets are sent). With this simple extension, our result on the problem would agree with DARPA's official verdict.

Feasibility Checking. We present our results on the STAC dataset without the feasbility checking discussed in Section 2.6. Performing symbolic execution on the STAC applications would require symbolically manipulating complex non-primitives, handling randomness, and symbolically executing non-trivial networking code. SPF currently does not provide such functionality, and manually constructing drivers by stubbing out method calls and simplifying non-primitive objects is beyond the scope of this paper. Instead, we use this benchmark to test CoCo-Channel's performance without its refinement stage. Importantly, neither of the false positives is a result of infeasible witness paths.

Optimization Queries. We provide the results of our optimization queries in Table 2.3. For

Table 2.2: Results on STAC Benchmark

Name	Size	Vulnerable	Correct	Time (s)	
				UC-CoCo	CoCo
collab	12	Safe	✓	0.026	0.052
gb1_mdp	90	Unsafe	✓	-	0.680
gb1_pwd	12	Safe	✓	0.042	0.090
gb2_mdp	86	Safe	×	-	0.857
gb3_search	286	Unsafe	✓	-	0.237
lawdb	736	Unsafe	✓	-	2.200
pb1_gen	83	Unsafe	✓	0.597	0.292
pb4_exp	94	Unsafe	✓	0.063	0.105
pb4_gen	18	Safe	✓	0.087	0.097
sb2_mdp	71	Unsafe	✓	-	0.730
sb3_pwd	13	Safe	✓	0.030	0.078
wm4_connect	21	Unsafe	✓	1.220	1.920
wm4_exp	901	Safe	✓	0.002	0.002
wm5_connect	916	Safe	×	35.200	2.100

Table 2.3: Optimization Results on STAC Benchmark

Name	#loop_iteration	Maximum Threshold	Time (s)
collab	-	19	0.031
gb1_pwd	37	506	0.061
pb1_gen	2	11853	1.280
pb4_exp	10	2052	0.087
pb4_gen	334	57	0.115
sb3_pwd	34	559	0.041
wm4_exp	-	0	0.002
wm4_connect	4	3688	1.240

unsafe variants, we report the results of the optimization queries on the smallest vulnerable component. For safe variants, we report the results on the largest component containing a secret-dependent component. Our experiments demonstrate that these queries scaled in eight of our examples. In two such benchmarks (**withmi4_exp** and **collab**), no loop was involved in a secret-dependent component, making the loop optimization query trivial. On the remaining six (**lawdb**, **gabfeed3_search**, **withmi5_connect**, and all the **_mdp** examples), Z3 either ran out of memory or was terminated after an hour. In the non-vulnerable cases to which the analysis scaled, it confirmed our suspicions of non-vulnerability by specifying the relatively large number of loop iterations necessary in order to introduce a side-channel or the relatively low maximum cost difference with our given loop bound.

2.7.2 Comparison to the State of the Art

We also evaluate CoCo-Channel against two recent tools for detecting side-channel vulnerabilites in Java applications, Blazer [19] and Themis [20]. Neither are publically available, so we were unable to test them on our STAC dataset. Instead, we compare CoCo-Channel on the benchmark introduced by the Blazer authors [19] and later used to evaluate Themis [20] as well as the additional benchmarks collected by the Themis authors [20].

Blazer Dataset

The Blazer dataset consists of 22 benchmarks drawn from a combination of challenge programs from the DARPA STAC program, classic examples from the literature [21–23], and microbenchmarks constructed by the Blazer authors. For our analysis, we used a loop bound of 10 and a threshold of 50. Originally, we inteded to choose parameters to match those in [20]; howver we We summarize the results of our experiments in Table 2.4. CoCo-Channel was able to correctly verify every program in the benchmark. Each tool was run on different hardware, making the timing reported incomparable, but CoCo-Channel's running time appears to be similar of Themis without compositional solving and faster than both tools when compositional solving is enabled. The benefits of compositional solving are not apparent for the smaller programs but are for larger ones.

Importance of Feasibility Checking. Without both kinds of feasibility analyses, CoCo-Channel would have incorrectly generated false positives and marked a safe variant as vulnerable. Our running time includes these analyses. The single path feasibilty analysis is performed any time a pair of witness paths of paths are returned by CoCo-Channel while the relational feasbility analysis is only performed when the cost differential is due to choices on secret-independent branches inside a secret-dependent component across both witness paths.

Optimization Queries. Table 2.5 shows the results of our optimzation queries, which scaled to

all programs except the unsafe **ModPow** variants. Since our choices of threshold and loop bound may be somewhat arbitrary, these provide insight into the degrees of vulnerablity in the programs beyond a binary classification. For example, the results on **array_safe** reveal that the program is very resilient to timing side-channels even for an adversary with refined observational abilities. Our threshold maximization results contradict the results reported in [20], where the authors claim that THEMIS is able to correctly determine the safety of the variants for a threshold of 0. We contacted the authors about this discrepancy, and they concede that it is likely due to a bug in their tool. While the exact value reported by the maximization query is dependent on the cost model, there is indeed some cost difference present in many of the safe variants.

Assumptions. In some of the safe variants (**k96**, **gpt14**, **modPow1**, **modPow2**), there is a conditional related to the length of the secret that leads to information leakage. Initially, our taint analysis correctly marked this conditional as secret-dependent and CoCo-Channel determined that the variant was vulnerable. Strictly speaking, there is a side-channel in these variants, and we consulted the authors of THEMIS about the discrepancy. They agreed and mentioned that since side-channels that only leak length are not considered strong enough, they manually annotated certain unsafe conditionals to tell THEMIS to ignore them. After doing the same, we re-ran the experiments and obtained the results reported.

THEMIS Dataset

The authors of THEMIS also evaluated their tool on a benchmark of real world Java programs with known vulnerabilities [20]. We compare our results on those benchmarks containing potential timing side-channels in Table 6. We set our threshold to 64 to match the threshold chosen in [20] and our loop bound to 10. Results are given in Tables 2.6 and 2.7.

CFG extraction. Janalyzer requires a jar file containing all relevent classes for control flow graph extraction. In many cases, the manually modified source code provided by the THEMIS authors was uncompilable due to missing dependencies or compilation errors. In these cases,

Table 2.4: Results on Blazer Benchmark

Name	Version	Size	Time (s)				Feas.	Rel.
			Blazer	Themis	UC-CoCo	CoCo		
MicroBench								
array	Safe	16	1.60	0.28	0.446	0.450	×	✓
	Unsafe	14	0.16	0.23	0.426	0.434	✓	×
loopAndbranch	Safe	15	0.23	0.33	0.968	0.982	✓	✓
	Unsafe	15	0.65	0.16	1.228	1.238	✓	✓
nosecret	Safe	7	0.35	0.20	0.001	0.001	×	×
	Unsafe	9	0.28	0.12	0.435	0.447	✓	×
sanity	Safe	10	0.63	0.41	0.624	0.623	×	✓
	Unsafe	9	0.30	0.17	0.439	0.450	✓	×
straightline	Safe	7	0.21	0.49	0.024	0.034	×	×
	Unsafe	7	22.20	5.30	0.413	0.423	✓	×
STAC								
modPow1	Safe	18	1.47	0.61	0.027	0.038	×	×
	Unsafe	58	218.54	14.16	18.016	0.597	✓	×
modPow2	Safe	20	1.62	0.75	0.028	0.038	×	×
	Unsafe	106	7813.68	141.36	181.412	0.675	✓	×
passwordEq	Safe	16	2.70	1.10	0.038	0.063	×	×
	Unsafe	15	1.30	0.39	0.494	0.519	✓	×
Literature								
k96	Safe	17	0.70	0.61	0.290	0.038	×	×
	Unsafe	15	1.29	0.54	0.411	0.424	✓	×
gpt14	Safe	15	1.43	0.46	0.028	0.030	×	×
	Unsafe	26	219.30	1.25	0.535	0.605	✓	×
login	Safe	16	1.77	0.54	0.07	0.089	×	×
	Unsafe	11	1.79	0.70	0.414	0.428	✓	×

Table 2.5: Optimization Results on Blazer Benchmark

Name	Version	#loop_iteration	Maximum Threshold	Time (s)
MicroBench				
array	safe	-	1	0.041
	unsafe	5	112	0.034
loopAndbranch	safe	21	28	0.116
	unsafe	5	98	0.054
nosecret	safe	-	0	0.001
	unsafe	6	88	0.020
sanity	safe	-	0	0.040
	unsafe	6	81	0.035
straightline	safe	-	5	0.030
	unsafe	-	1343	0.030
STAC				
modpow1	safe	51	9	0.035
modPow2	safe	51	9	0.035
passwordEq	safe	51	9	0.052
	unsafe	4	144	0.036
Literature				
k96	safe	17	27	0.037
	unsafe	-	74	0.020
gpt14	safe	26	18	0.037
	unsafe	4	175	0.069
login	safe	13	36	0.090
	unsafe	4	168	0.022

we manually wrote a driver for the provided source code and stubbed out any method calls we were unable to find the dependencies for. For one example, **pac4j**, stubbing out methods was a prohibitive process so we removed this example from our evaluation. In the other cases, we believe the integrity of the benchmark to remain intact but note that the subgraph of the control flow graph we analyze may differ from that analyzed in [20].

Disrepancies. CoCo-Channel incorrectly determines that the safe variant of **tomcat** is vulnerable. In the Themis dataset, the authors manually patch the vulnerable **tomcat** variant by introducing an expensive method call into the shorter branch to mitigate the vulnerability. The effects of this patch can be seen in the results of our threshold maxmization query across the two versions. Nevertheless, CoCo-Channel reports that it is not robust enough to fully hide the side-channel. This could be due to a difference in our cost model, our method stubs not providing the same time complexity as the original code, or a different assumption on the part of the Themis authors on what paths are feasible. We also note that we did not perform our

Table 2.6: Results on Themis Benchmark

Name	Version	Correct	Time (s) Themis	Time (s) UC-CoCo	Time (s) CoCo	Feas.	Rel.
Spring-Security	Safe	✓	1.70	0.034	0.068	×	×
	Unsafe	✓	1.09	0.831	0.851	✓	×
JDK7-MsgDigest	Safe	✓	1.27	0.015	0.028	×	×
	Unsafe	✓	1.33	0.459	0.473	✓	×
Picketbox	Safe	✓	1.79	0.015	0.044	×	×
	Unsafe	✓	1.55	2.889	0.589	✓	×
Tomcat	Safe	×	9.93	1.280	1.010	×	✓
	Unsafe	✓	8.64	1.280	0.950	×	✓
Jetty	Safe	✓	2.50	0.038	0.201	×	×
	Unsafe	✓	2.07	0.480	1.531	✓	×
orientdb	Safe	✓	37.99	0.178	0.406	×	×
	Unsafe	✓	38.09	2.982	0.842	✓	×
boot-auth	Safe	✓	9.12	0.018	0.060	×	×
	Unsafe	✓	8.31	0.443	0.461	✓	×

single path feasbility analysis on **tomcat** as, like the STAC benchmark, doing so would require symbolically manipulating non-primitives and extensive stubbing of method calls. Nevertheless, we manually confirmed that this false positive was not due to the infeasibility of the returned paths.

Assumptions. As in the Blazer examples, many of the safe variants actually do contain timing side-channels that leak the length of the secret or if the secret is null. Following the example of the Themis authors, we manually mark secret-dependent conditionals that depend only on the length of the secret, whether the secret is null, or if it is valid input as "safe". Exceptions are made for a length check in **spring-security** and for a null check in **tomcat** in which the intended vulnerabilities are side-channels of such kind.

2.8 Related Work

Side-Channel Detection. Work on side-channel detection constitutes a large field of research. The work most immediately to mine is that of Antopolous et al. [19] and of Chen et al. [20]. Their tools Blazer and Themis are both motivated by the same understanding that secret-dependent differences in control flow can introduce side-channel behavior. Our technical contributions;

Table 2.7: Optimization Results on Themis Benchmark

Name	Version	#loop_iteration	Maximum Threshold	Time (s)
Spring-Security	safe	-	0	0.041
	unsafe	3	179	0.061
JDK7-MsgDigest	safe	-	0	0.018
	unsafe	5	136	0.023
Picketbox	safe	-	1	0.018
	unsafe	4	208	2.980
Tomcat	safe	1	275	0.610
	unsafe	1	350	53.600
Jetty	safe	-	1	0.044
	unsafe	4	233	0.050
orientdb	safe	-	1	0.198
	unsafe	4	208	2.950
boot-auth	safe	-	36	0.020
	unsafe	5	144	0.026

however, differ. Blazer uses a grammar-based approach to representing program trails while Themis introduces a simple imperative language with cost-instrumented operational semantics and verifies programs in this language using a relational program logic. Our compositional annotation and solving ability provides my approach with additional scalability.Additionally, I use symbolic execution to check the feasibility of our results while both Blazer and Themis use abstract interpretation, though my relational feasibility analysis shares similarities with Themis's loop invariant generation algorithm.

Another approach is taken by Pasareanu et al. [23], Bang et al. [24], and Phan et al. [25]. Their work uses a combination of symbolic execution and model counting to determine the probability of a particular execution path and tools from information theory to quantify the leakage based on those probabilities. Due to its reliance on full symbolic execution, this approach does not scale to larger applications. My approach can be considered complementary to theirs. Using CoCo-Channel, we can reduce a complicated program to a potentially much smaller set of suspicious components on which their analysis might scale.

Zhang et al. on Sidebuster [26] detail a static approach to detecting side-channels in web applications. Their approach, like mine, involves determining the cost difference between paths resultant from secret-dependent conditionals. My use of symbolic variables in cost expres-

sions allow for a significantly more expressive and refined analysis than theirs which tracks and compares the remote procedure calls in each branch. Additional static approaches to side-channel detection include type-based approaches [27, 28] or are particular to specific types of side-channels, such as CacheAudit [29]. Dynamic approaches to side-channel detection [30, 31] is also an area of active research. DiffFuzz [32] is a differential fuzzing technique aimed towards finding program executions that maximize cost difference and is an example of a dynamic strategy aimed towards the same goal as CoCo-Channel.

Path-Based Worst-Case Analysis. My work is similar to research aimed at determining the worst-case execution time of a program [33]. Li and Malik [34] developed an approach which, like mine, implicitly considers all program paths. They determine a cost per basic block, introduce variables denoting the number of traversals of each basic block augmented by structural constraints, and then maximize the sum of the product of the costs of each basic block and its number of iterations. Puschner and Schedel [35] similarly aim at computing the worst case computation of a program and exploit the visual nature of the control flow graph to develop a set of constraints describing possible paths through the control flow graph. Though both works share the goal of implicit path enumeration, our choice of formulation differs and is dictated by our desire to detect not the maximum path through a program but rather the maximal secret-sensitive paths.

2.9 Chapter Summary

In this chapter, I presented a scalable static analysis for detecting side-channel vulnerabilities and locating the vulnerability-introducing aspects of code. CoCo-Channel provides compositional generation of symbolic path cost expressions, compositional query generation, optimization query formulation and evaluation, and symbolic execution based path feasibility analysis to eliminate false positives. I have demonstrated its ability to scale to large and

highly-realistic applications and obtain meaningful results on those systems.

Chapter 3

JITLeaks: Inducing Timing Side Channels through Just-In-Time Compilation

3.1 Introduction

The underlying assumption of side-channel analysis tools such as CoCo-Channel, Blazer and Themis is that the source code is the only determiner of the side-channel observable. This is not the case. Just-in-time (JIT) compilation, which is crucial to the performance of modern programming languages such as Java and Javascript, can introduce timing side channels into deceptively secure-looking code fragments when it attempts to optimize paths it deems "hot". In this chapter, I present a new class of side-channel vulnerabilities due to the optimizations introduced by just-in-time (JIT) compilation. I focus on the HotSpot JVM and Google V8 runtime engines, but any JIT-enhanced runtime is similarly susceptible.

I show that if the input distribution to a program is non-uniform, the JIT compiler state will be primed to favor certain paths, resulting in optimizations that reduce their execution time.

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

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

Figure 3.1: A naive password-checking method and a "fixed" one.

This can introduce timing side channels even in programs whose resource consumption has been carefully balanced.

3.2 An Overview of JIT-Based Side Channels

Consider the naive password-checking method shown in Fig. 3.1 (left). In this Java method, the password and a guess of matching length are compared character-wise. As soon as there is a mismatch, false is returned. This early return results in a timing side channel that enables an observer to correlate the execution time with the number of characters matched.

A security-conscious developer might decide that, since the method handles sensitive data, it is worth sacrificing the early return in exchange for a more secure function. They might propose a method like the one shown in Fig. 3.1 (right). In this new version of check, the same amount of work is performed regardless of the length of the matching prefix.

The side-channel vulnerability appears to have been fixed in the new version of the code. However, the source code written by the developer is not the only factor impacting the execution time of program paths. The runtime environment itself can introduce timing side channels into deceptively secure-looking code fragments when it attempts to optimize paths that it deems "hot." In this Java example, the HotSpot JVM tracks how often each branch of a conditional branch instruction is taken, and uses this information when JIT-compiling a method to generate native code favoring the more frequent branch. If an attacker is guessing potential passwords

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

Figure 3.2: Execution time of the "fixed" check method.

randomly, the probability of missing is much higher than that of matching something. As a result, the *then* branch heats up, and JIT introduces a timing side channel into the supposedly "fixed" version. Figure 3.2a depicts the clear separability of the method's execution time distributions when the first character misses the first character of the password (optimized branch) versus when it correctly matches it (non-optimized branch). Figure 3.2b shows how the side channel disappears when JIT is disabled.

The runtime behavior of the program introduces a side-channel vulnerability, enabling the attacker to learn whether the first character of the guess matches that of the password. This predicate is related to the branch condition in Fig. 3.1 (right). We present a guide describing how JIT-based side channels can enable an attacker to learn sensitive predicates on input and how to identify potentially vulnerable code. We assume that the attacker is interested in learning a predicate ϕ about any input from a certain subset of all possible inputs to program p. This subset is defined by the assumptions the attacker can make about the input (e.g., that the guess is the same length as the password as in the above example).

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

Figure 3.3: Attack models

3.2.1 Attack Models

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

Key to inducing and leveraging JIT-based timing channels is understanding that they arise from a bias in the distribution of inputs to the program. I refer to the act of interacting with a JIT-optimizing runtime in a biased manner as *priming* the runtime. Priming means repeatedly running a program with inputs that exercise certain paths, heating up the state of optimization in a way that favors those paths. A runtime can be primed in various ways, and how we assume it is primed greatly influences what kinds of JIT-based side channels may be used. Another key aspect is time measurement—what exactly do we assume that the attacker is able to time? Lastly, each attack model establishes the purpose of the attack—what does the attacker learn if she succeeds? Figure 3.3 summarizes the attack models whose details we present in the next sections.

3.2.2 Induced-Priming Model

The first attack model is the *induced-priming model* (IPM), in which we assume that the attacker is able to prime the runtime into a vulnerable state by repeatedly triggering the program p on an input value (or values) of her choice and is then able to time one subsequent call to p

made by another user with a secret value s. The attacker's goal is to determine whether s does or does not satisfy some predicate of interest ϕ. This model is only realistic in scenarios where we can assume that the attacker has dominant control of the runtime.

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

3.2.3 Natural-Priming Model

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

Typical Behavior

In the first version (NPM-LTB), the attacker aims to learn the typical behavior of the program. From the timing of her own probe $p(\pi)$, the attacker learns if her input π agrees or disagrees with the typical input to p with respect to the predicate ϕ. Imagine, for example, that there is an online referendum taking place. The attacker wants to know what decision is

favored by the majority. If enough users have voted disproportionately in favor of one decision, the runtime could have been primed to favor that choice. The timing of the attacker's probe $p(\pi)$ could thus leak information about whether her vote π represents the typical case.

Atypical Behavior

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

3.2.4 Roadmap and Contributions

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

1. **Priming:** p is executed repeatedly with an input distribution biased with respect to predicate ϕ.

2. **Timing:** The attacker times the execution of p for a particular input value.

3. **Inference:** Based on the observed execution time, the attacker infers the value of predicate ϕ on unknown input.

Imbalances introduced through biased behavior at runtime are related to various JIT optimizations. These optimizations interact in complex and subtle ways. Combined with noisy

timing, this makes the art of leveraging JIT-based timing side channels an intricate process. We identify several *vulnerability templates*, each based on exploitable JIT optimizations. These templates help us identify which predicates related to paths in p may be amenable to JIT-based vulnerabilities, and guide us in finding the right priming parameters or requirements.

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

I first apply my approach in a fine-grained analysis of Java and JavaScript methods. Since the timing distribution of different execution paths can overlap, we may not always reach full certainty about the value of the predicate, even if we induce a strong side channel. I use the conditional entropy between the timing information and the value of the predicate to quantify how much information about the predicate is leaked. I discuss the results and lessons learned from our most interesting experiments, both successful and unsuccessful.

I then experiment with the Apache Shiro [36] security framework and the GraphHopper [37] route planning server to explore how JIT-based side channels can be induced in large well-known applications. The results show that they can indeed be introduced, and that they can be sizable enough in magnitude to be observable over the public internet.

My contributions in this chapter are:

- Definition and demonstration of a new class of timing side channels due to JIT optimizations during runtime.
- Three attack models for learning predicates about secret inputs using JIT-based side channels.
- Five vulnerability templates to identify code fragments susceptible to JIT-based timing vulnerabilities.

- A profiling method to gather the statistical information needed to infer predicate values in noisy environments.
- Experimental evaluation of applying our approach to widely used Java and JavaScript functions.
- Examples and experimental analysis of multiple JIT-based side channels in two well-known Java frameworks.

The rest of the chapter is organized as follows: In Sect. 3.3 I review JIT optimizations and introduce related vulnerability templates. In Sect. 3.4 I present algorithms to effectively use timing information arising from JIT-based side channels. In Sect. 3.5 I describe our experiments on built-in Java and JavaScript functions. In Sect. 3.6 I discuss the experimental results. In Sect. 3.7 I demonstrate JIT-based side channels in well-known frameworks. In Sect. 3.8 I discuss related work. In Sect. 3.9 I present our conclusions and ideas for future work.

3.3 Vulnerability templates for JIT-based side-channels

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

3.3.1 Just-in-time (JIT) compilation

Just-in-time compilation has been used since 1960 [38] to improve the performance of interpreted languages. Based on the observation that a majority of execution time is typically spent executing a small fraction of the code [39], runtime profiling can be used to detect "hot" functions or code portions that are worth feeding into an optimizing compiler. This involves a complex trade-off between initial compilation delays ("warm-up time") and subsequent performance benefits.

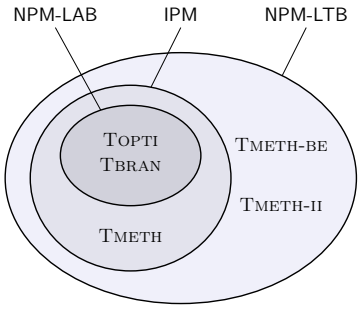


Figure 3.4: Vulnerability templates for each attack model.

Modern JIT compiler implementations involve techniques to dynamically adjust the optimization level (and thus the compilation overhead) of each method in order to maximize the return on investment. Aggressive speculative optimizations may give rise to the need for recurrent assumption checks, and to deoptimization penalties when such a check fails [40].

JIT compilation has been employed by many languages including LISP [41], Smalltalk [42], FORTRAN [43], APL [44], PHP [45], Java [46], and JavaScript [47].

3.3.2 Vulnerability Templates and JIT Compilation Techniques

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

Each vulnerability template has:

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

- The attack model(s) for which the template is harnessable.

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

Branch prediction (TBRAN)

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

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

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

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

Optimistic compilation (TOPTI)

If, when a method is aggressively compiled (e.g. elevated to a higher tier of compilation in HotSpot), the counters shows that one side of a conditional is *very* rare, the optimizing compiler may make the optimistic assumption that the branch will never be taken. Similarly, if counters

show that a potentially polymorphic call site always calls the same receivers, the optimizing compiler may assume the rarely-seen dispatches never occur. In either scenario, if and when the rare case occurs and the optimistic assumption is broken, the method must be *de-optimized* and replaced with a slower, more conservative version.

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

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

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

Method compilation (TMETH)

Compilation decisions are impacted by runtime profiling metrics. One key metric is method invocation count. When the number of method invocations reaches a certain threshold [48], the method may be scheduled for compilation or more aggressive recompilation. Another factor that may promote (re)compilation of a method are back-edge counters that track how often backward jumps (typically due to loops) are taken. TMETH exploits the speed difference between interpreted and compiled (or between conservatively and aggressively compiled) code.

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

Recipe Priming must ensure that m is executed a sufficiently high number of times, so that m is compiled to a faster version. The speeding up of m thus causes or augments an observable imbalance in the timing of p.

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

Method compilation due to back-edges (TMETH-BE)

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

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

Recipe The priming amount must be in the range to induce a difference between the optimization level of m according to the two priming scenarios. The ideal probe value for this vulnerability template is one for which the method m is expensive—making the difference in execution time between its differently compiled versions more apparent.

Attack Models NPM-LTB.

Method compilation due to imbalanced invocations (TMETH-II)

This template is specific to NPM-LTB.

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

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

Attack Models NPM-LAB.

3.4 Statistical Profiling for Accurate Inference

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

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

3.4.1 Learning under IPM

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

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

Algorithm 1: Prime pseudocode.

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

Algorithm 2: IPM attack pseudocode

more distinguishable the profiles under the two cases, the more successfully the attacker has booby-trapped the runtime engine.

Obtaining a statistical profile benefits us twofold. First, time measurements are affected by nondeterminism from various sources, from inevitable system noise to minor variations in runtime decisions made by the JIT compiler as to which optimizations to apply and in what order. The statistical nature of the profile accounts for such noise. Second, the assumption that the attacker has complete control over the runtime is often unrealistic. When we build a statistical profile, we can simulate an environment where some proportion of calls to p is outside

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

Algorithm 3: NPM-LTB attack pseudocode

the control of the attacker. By *priming with an α distribution* (with respect to ϕ) we mean priming p with inputs satisfying ϕ with probability α, and $\neg\phi$ with probability $1 - \alpha$. When we build a statistical profile, we prime with $\alpha < 1$ to simulate a context where p is occasionally triggered on inputs satisfying the opposite value to the one we have chosen to heat up.

The pseudocode in Algorithm 2 outlines the above process. Here, the two priming input values $p_\phi, p_{\neg\phi}$ are chosen such that (a) both satisfy any assumptions the attacker makes over the input space, and (b) p_ϕ satisfies ϕ whereas $p_{\neg\phi}$ satisfies $\neg\phi$. The test values t_ϕ and $t_{\neg\phi}$ are chosen randomly from the set of possible secret values (T_ϕ and $T_{\neg\phi}$) satisfying ϕ and $\neg\phi$ respectively (along with any additional assumptions on the input space) to generate representative timing information for a secret value. The priming amount n is the total number of calls to the program p in the Prime algorithm (see Algorithm 5) and the profiling amount N is the number of times the priming and then timing subroutine is repeated during profiling in order to generate a statistical profile robust to noise.

Choice of test values can impact the accuracy of the statistical profile. The more similar the test values t_ϕ and $t_{\neg\phi}$ to the actual unknown value, the more accurate the profile likely is. Here similarity means that modulo branch decisions correlated with the predicate, the test input

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

Algorithm 4: NPM-LAB attack pseudocode

follows a similar program path as the unknown input. Since this cannot be known beforehand, the attacker's best option is to generate and profile for a wide set of test values. Likewise, choice of priming input can impact how successful an attacker is in booby-trapping the runtime. Ideally the attacker would choose priming input following the same program path as the unknown input. If this is not possible, an attacker could vary her priming input over a set of possible paths in an attempt to avoid introducing a timing channel due to an entirely different predicate.

3.4.2 Learning under NPM

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

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

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

3.5 Library Evaluation: Setup

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

3.5.1 Source of experimental subjects

I evaluated this approach on the **java.math.BigInteger**, **java.lang.Math** and **java.lang.String** classes from the Java standard library (JDK 8, rev. b132) [49] and on the **Math**, **String** and **Array** JavaScript objects from Google's V8 open-source JavaScript engine (V8 4.5.103.35) [50]. I removed methods with no conditionals, native methods not written in Java or JavaScript, du-

Table 3.1: Priming distributions used in our experiments

	IPM		NPM-LTB		NPM-LAB	
	Java	JS	Java	JS	Java	JS
Topti	0.998	1.000	0.998	1.000	0.998	1.000
Tmeth	0.950	0.950	0.950	0.950	n/a	n/a
Tbran	0.900	0.900	0.950	0.950	0.900	0.900
Tmeth-be	n/a	n/a	0.950	0.950	n/a	n/a
Tmeth-ii	n/a	n/a	0.950	0.950	n/a	n/a

plicates modulo type (e.g., float vs. double), and those with isomorphic control-flow structures. For the remaining methods we applied our approach and chose predicates for conditionals that satisfied the most relevant templates. In the following sections I show a selection of my results featuring the cases (successful and unsuccessful) that I found most interesting and relevant.

3.5.2 Computing information leakage via conditional entropy

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

3.5.3 Using priming distributions to simulate noisy triggering

As discussed in Section 3.4, the α ratio accounts for the fact that in a realistic scenario, we will not have exclusive control over the state of the runtime engine and the bias under NPM may not be absolute. Table 3.1 shows the distributions that we associated with each template under each model for both Java and JavaScript programs.

3.5.4 IPM experiments

For each case under IPM, we chose two values for priming: one satisfying ϕ and one satisfying $\neg\phi$, following the approach given in Algorithm 2. We then generated two sets of possible secret inputs satisfying ϕ or $\neg\phi$ respectively and both satisfying a set of additional assumptions over the space of all possible inputs. These assumptions are further discussed in Section **??**. We primed the runtime engine with the priming values using the priming ratio α indicated by the template.

For TMETH and TOPTI cases we determine the number of priming iterations as follows: Starting with an initial guess, use the JITWatch tool [51] for Java, or the `--trace-opt` option for JavaScript, to determine whether the optimization has taken place. If not, increase the number of iterations until it does. For TBRAN cases we tried priming {1000, 10000, 50000, 100000} times and kept the value that maximizes leakage.

For evaluation, I repeated the following 1000 times. I primed the runtime engine using one priming value as described above, then timed a call to the method on a randomly chosen secret value satisfying ϕ. Then I performed another 1000 iterations of the experiment, now timing a call to the method on a randomly chosen secret value satisfying $\neg\phi$. From this data I computed the leakage as explained in 3.5.2.

In addition, we also re-executed all experiments (and leakage computation) for the following three priming scenarios:

Reversed priming

I re-ran all experiments with a ratio $\overline{\alpha} = (1 - \alpha)$ instead of α. In other words, if the runtime was primed more heavily favoring ϕ in the original experiment, it is now primed more heavily with input satisfying $\neg\phi$, and vice versa. This evaluates whether that test subject is reversible.

Even priming

I re-ran all experiments with a fixed ratio $\alpha = 0.5$, i.e., the amounts of priming satisfying ϕ and $\neg\phi$ are equal. This evaluates the importance of the imbalanced priming ratio in introducing a side channel, as opposed to the more general, overall heating up of the whole method.

No JIT

I re-ran all experiments with JIT disabled. This evaluates the existence of a static (traditional, source-code level) side-channel vulnerability, which our use of JIT could augment or mitigate. We still used a very small, fixed, balanced amount of priming (50 calls on both sides) to avoid artificial noise from initial class/method loading delays.

3.5.5 NPM experiments

For cases evaluated under NPM, I generated two sets of possible priming values, one satisfying ϕ and the other $\neg\phi$. For each case, we determined the number of priming iterations in the same way as for IPM (see 3.5.4), using JITWatch or `--trace-opt` to guide the search.

NPM-LTB experiments For evaluation, I repeated the following experiment 1000 times. I primed the runtime engine (with priming parameters obtained as described above) in favor of priming values satisfying ϕ, and then timed a subsequent call on a chosen probe input π. I manually chose π such that the difference between the optimization levels after the two types of priming would be observable. I experimented again by priming the runtime engine with the same parameters as before, but in favor of the priming inputs satisfying $\neg\phi$, and then timed a subsequent call on the same probe π. Each experiment was repeated 1000 times. From this data I computed the leakage as explained in 3.5.2. Since NPM-LTB is the most expressive model in terms of the vulnerability templates applicable under it, we focus our experiments on methods and predicates satisfying templates un-harnessable under the other models.

NPM-LAB experiments I additionally generated two sets of possible secret inputs satisfying ϕ or $\neg\phi$ respectively and both satisfying a set of additional assumptions over the space of all possible inputs. For evaluation, we did the following 1000 times. I primed the runtime (with priming parameters obtained as above) in favor of the priming values satisfying ϕ, then made a call to the method on a randomly chosen secret value executing the $\neg\phi$ branch. I then timed a subsequent call on a chosen probe input π. Then I performed another 1000 iterations of the experiment, this time calling the method on a randomly chosen secret value executing the ϕ branch and timing the subsequent call on the same probe input π. From this data we computed the leakage as explained in 3.5.2. I then re-ran all experiments under the reverse priming model described in 3.5.4. This evaluates whether the natural priming needs to be more strongly in favor of one particular value of ϕ for atypical behavior to be detected.

3.5.6 Hardware setup

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

3.6 Experimental results

Tables 3.2, 3.3, and 3.4 summarize our results for Java and JavaScript methods under IPM, NPM-LTB, NPM-LAB respectively. For each set of experiments we report the method name, location of the selected branch instruction in the class source code [49,50], the template applied, other templates (if any) that also arose unexpectedly, and the priming parameters used. In all cases, we report the amount of information leaked about the predicate value under all evaluated priming scenarios.

Table 3.2: Experimental results for IPM in Java (top) and JavaScript (bottom)

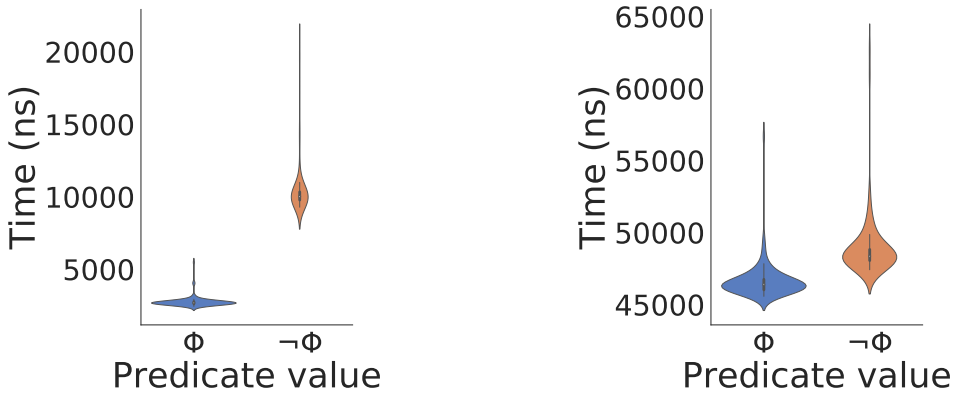
Programing language	Method name	Branch instruction	Template (applied, arisen)	Priming amount	Priming ratio (α)	Leakage under α	Leakage under $\overline{\alpha}$	Leakage 0.5 | 0.5	Leakage w/o JIT
Java	BigInteger.min	line 3477	Topti	100,000	0.998	**1.00**	**1.00**	0.02	0.06
Java	BigInteger.valueOf	line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

Table 3.3: Experimental results for NPM-LTB in Java (top) and JavaScript (bottom)

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

Table 3.4: Experimental results for NPM-LAB in Java (top) and JavaScript (bottom)

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



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

Figure 3.5: Execution time distributions for JavaScript method Math.max under (a) IPM and (b) NPM-LAB.

3.6.1 Optimistic compilation (TOPTI)

When optimistic compilation could be induced, a sizable timing difference arises (e.g., see Fig. 3.6b), resulting in very reliable learning of $\phi(s)$ under IPM. Our high-leakage results for Java methods BigInteger.min, Math.nextAfter, and String.compareTo and JavaScript methods Math.max, Math.aSinh, String.charAt, Array.toLocaleString and Array.lastIndexOf were obtained in this way. Leakage is also reliably high for these methods under NPM-LAB. When the call on the unknown value breaks the optimistic assumption, the runtime engine must revert to a less optimized version of the method under test. This results in an observable difference in the timing of the attacker's probe to the method when compared to the case where the optimistic assumption is not broken (and the highly optimized code used). This timing difference can be augmented by choosing a probe value for which the method is expensive and the difference between versions more apparent. Nevertheless, the magnitude of the timing difference is less than in IPM when the actual execution time of the call breaking the optimistic assumption is measured. Figure 3.5 demonstrates the relative difference in magnitude across the two attack models for the JavaScript method Math.max.

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

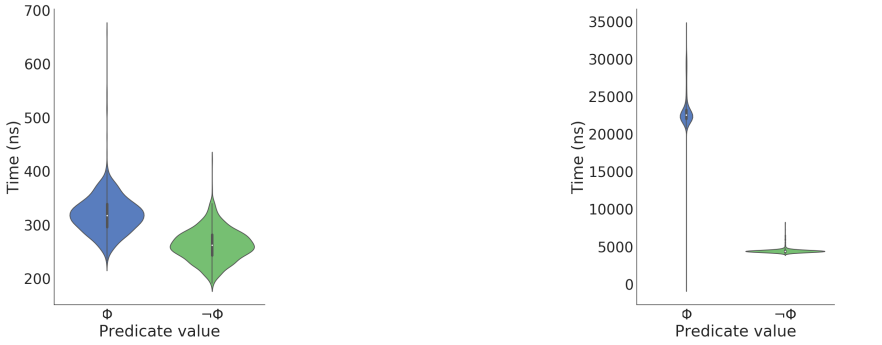
For the other two Java cases, Math.min and String.equals and the JavaScript case Array.slice, our priming did not succeed in inducing optimistic compilation. In Math.min and Array.slice, this was because the functions themselves were not compiled despite their large number of invocations. String.equals was compiled but we could not induce optimistic compilation due to

a combination of two facts: (i) optimistic compilation requires an extremely lopsided history at the time of aggressive compilation, and (ii) String.equals is triggered too frequently by other parts of our experiment driver. Hence, this template is suitable for contexts where the attacker has nearly-exclusive control over the triggering of p. In contrast, the String.compareTo method, which has an almost identical structure to String.equals with respect to the selected predicate and its branches, was much more amenable to leakage from optimistic compilation due to its less frequent usage elsewhere.

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

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

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



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

Figure 3.6: Execution time distributions after priming for Java methods (a) String.equals and (b) String.compareTo.

faster.

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

3.6.2 Method Compilation (TMETH)

Our high-leakage result for BigInteger.shiftLeft exemplifies the potential of TMETH under IPM. In BigInteger.shiftLeft, a different method is called depending on the value of ϕ. With JIT disabled, the execution time *does* leak information about which branch was taken. This is not surprising, as one can expect that the unoptimized versions of two different methods would be distinguishable. What we wish to emphasize is that any of the two callee methods can be made observably faster than the other through the appropriate priming. Moreover, both priming scenarios result in stronger side channels than those that occur with JIT disabled or with an

even priming distribution. This demonstrates how strongly the execution time of a path can vary depending on how aggressively the methods called along that path are optimized.

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

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

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

The difference between the two results for Java function BigInteger.mod stems from an experiment allowing the attacker stronger timing abilities. The second row gives the leakage

when we do not time p, but rather its callee m whose compilation we aim to induce. Such refined timing information substantially increases leakage, but requires more assumptions.

3.6.3 Branch Prediction (TBRAN)

Branch prediction introduces considerably smaller timing differences than other templates (e.g., see Fig. 3.6a). Nevertheless, it can still sometimes be exploited to great effect. The Java methods BigInteger.valueOf, String.startsWith, and Math.max and JavaScript method String.indexOf are examples of methods that are small enough that the effect of branch prediction is observable over the computational noise of the method. JavaScript method Math.trunc, on the other hand, is an example where the timing difference is too small to be reliable. Whether or not the branch condition is looped over can also impact the observability of the side channel. In NPM-LTB, where we can choose the test value, a looping construct may enable the choice of a test value for which the effects of branch prediction are multiplied, i.e., the branch prediction is repeatedly correct or incorrect across iterations of the loop. String.replace (discussed in the next section due to its interaction with TMETH-BE) is an example of this scenario.

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

3.6.4 Method Compilation via Back Edges (TMETH-BE)

This template is specific to NPM-LTB. Every time we tried to apply TMETH-BE, we successfully found priming amounts such that the method was compiled to different levels of optimization. In the Java cases BigInteger.and and String.trim cases and the JavaScript cases Math.min, String.concat, and Array.map, we were able to find a probe value expensive enough for differing levels of compilation to be observable. This is due to the large number of potential loop iterations within these methods. This was not the case in the Java method Math.scalb, where the maximum possible number of loop iterations is four. The strength of a side channel introduced by TMETH-BE is thus bounded by how expensive the method in question can be made by suitably chosen probe values.

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

3.7 Application leakage examples

3.7.1 Apache Shiro

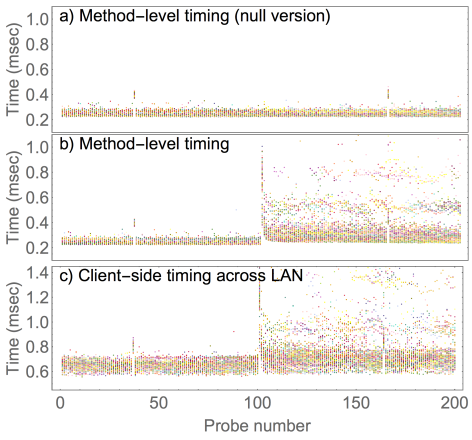
Apache Shiro [36] is an easy-to-use, open-source Java security framework for authentication. It has over 2000 stars on Github as of this writing. Developers use Shiro to add permissions, roles, and session management to their applications.

Shiro Tutorial

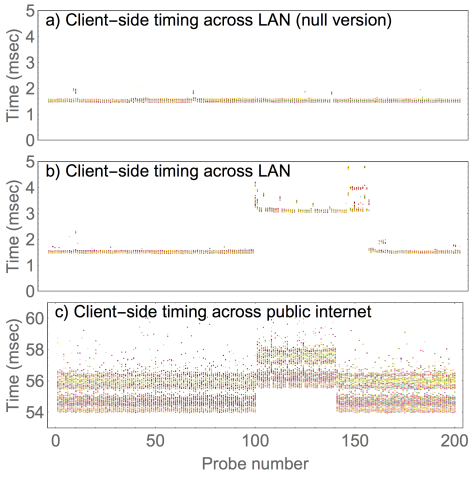
The official tutorial shows how to integrate Shiro into your application using a simple user database. Given a username and password, the example code performs a Shiro login with the given credentials, checks them against the database, tests whether the user has a permission, and reports whether they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

We experimentally demonstrate. Using the unmodified Shiro tutorial code inside a loop, we make unprivileged users prime the system 5000 times by repeatedly logging in and consequently heating up the typical branch; an unprivileged attacker probe the system 200 times through her own login; and a privileged user log in between the attacker's 100'th and 101'st probes. Due to JIT nondeterminism, we repeat the experiment 100 times. Figure 3.7ab shows the 100 superimposed traces. The point at which the atypical event happens is clearly visible: the attacker's 101'st probe (first one after the event) takes an unprecedented amount of time. The following probes are also more expensive, although the effect soon wears down.

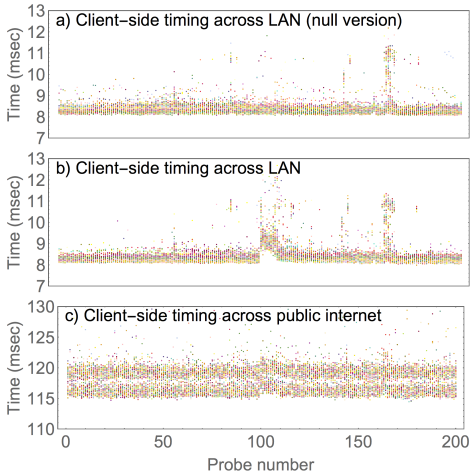
In Figure 3.7aa I show the null version of the experiment: same conditions, priming, and probing, but the atypical event is replaced with a typical one. This aims at confirming that the



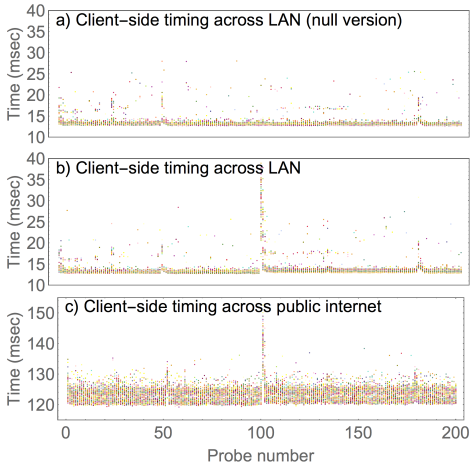
(a) Apache Shiro: Tutorial example (unmodified).



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



(c) GraphHopper: Maximum distance vulnerability.



(d) GraphHopper: Atypical algorithm vulnerability.

Figure 3.7: Plots for Apache Shiro and GraphHopper experiments

phenomenon is caused by the presence of the atypical event, rather than some other aspect of our experimental setup.

We then wrap the same Shiro tutorial program in a simple TCP server. A TCP client connects to the server from a different computer and issues login/action/logout commands. The same priming, probing, and atypical event as before are now executed on the server at the client's requests. Response times are measured on the client side. The LAN setup is described in Section 3.7.3. Figure 3.7ac shows that, even partially fuzzed by network noise, the phenomenon is still clearly observable through our LAN. However, it is not strong enough to be realistically observable through the public Internet.

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

Amplification through computation

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

We perform a similar experiment as before. We can use fewer priming iterations (1000)

since additional back-edges cause the function to compile earlier. Figure 3.7ba shows the null experiment. Figure 3.7bb shows the results when the atypical event occurs after the 100'th probe. Figure 3.7bc shows that the effects of reverting to less optimized code are observable over the public Internet, and last until the method is recompiled. This increased observability is due to the now larger difference between the highly optimized and less optimized versions of the function. While we experimented with a specific non-trivial, highly-optimizable function, any method involving computation satisfying similar properties and containing a Shiro permission check would be similarly susceptible.

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

3.7.2 GraphHopper

GraphHopper [37] (GH) is an open-source framework that computes directions on city maps. It uses maps from the OpenStreetMap [52] project. The GH server can answer queries like "best route from A to B by train in Berlin" issued by clients through a RESTful API. GraphHopper is a well-known project with over 1,800 stars on Github as of this writing.

We present two examples of leakage due to optimistic compilation in GH. One allows an

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

We did not modify GraphHopper in any way. Our experiments can be replicated using the unmodified current distribution of the GH server (3.7.3) and the map of Berlin.

Distance Threshold

GH has a configurable maximum separation (graph edges) between allowable *from* and *to* points. We used a limit of 5000 edges. We experimented under the assumption that the majority of users issued queries within range. We collected 100 traces of the following experiment. We primed the JVM with 3000 routing queries between random locations within range; probed with a routing query between two fixed locations within range; and made a routing query between two random locations outside of the range between the 100'th and 101'st probe. Figure 3.7cb shows the results over the LAN. Figure 3.7ca shows the null version. Figure 3.7cc shows the results over the public Internet. Though not perfectly reliably, this timing channel is observable over the public Internet and the attacker is likely able to infer if and when a user makes an atypical routing query.

Routing Algorithm

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

We collected 100 traces of the following experiment. We primed the JVM with 1000 routing

queries between random locations using Dijkstra's algorithm; probed with a routing query between two fixed locations using Dijkstra's algorithm; and made a routing query on two random locations using the A* algorithm between the 100'th and 101'st probe. Figure 3.7db shows the results over a LAN. Figure 3.7da shows the null version. Figure 3.7dc shows the results across the public Internet. The shift in the timing of the probes is observable over the public Internet. In fact, the probe following the rare behavior takes much longer than any prior probe (usually by ~15 msec) due to the less compiled version of the relevant routing-algorithm-handling code being noticeable more expensive for probes requiring many iterations of the algorithm. However, using such an expensive probe means many back edges are taken, resulting in quick recompilation and a fading effect.

3.7.3 Experimental setup

I ran Apache Shiro v1.3.2 and GraphHopper v11.0 on two Intel NUC 5i5RYH computers. Both machines are on our Ethernet LAN via a Netgear GS108Ev3 switch. Another five computers are on the LAN. Under low load, typical round-trip time between the client and the server machines through the LAN was 0.27 msec (min 0.25, mean 0.273, max 0.34).

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

3.8 Related work

To the best of my knowledge, the idea that JIT could impact and potentially introduce timing channel vulnerabilities was first put forth by Page [53]. Noting that compiled code can

differ from source code, he explores the impact of dynamic compilation through a case study on his own Java implementation of a double-and-add-based multiplication program. Because the doubling method is called more frequently than the addition method, it is compiled sooner. If an attacker can obtain a timing profile of each method called within the multiplication code, she can infer the order of the sequence of doublings and additions performed. Page also proposes some potential mitigations at both the language level and the virtual machine level. My work goes beyond the observation that dynamic compilation *may* introduce side channels by demonstrating how to systematically induce JIT-based runtime-behavior-dependent side channels through bias in input distribution. I *actively* exploit JIT's focus on optimization to create side channels and demonstrate our approach on real applications.

In work complementary to mine, Cleemput et al. [54] propose leveraging the profiling information used in dynamic compilation to mitigate timing side channels. Starting from a developer-chosen root method, profiling information on the number of back edges taken or method invocations is collected for values in a training input set. Based on this process, a set of methods potentially vulnerable to timing channels is selected. Control-flow and data-flow transformations are then applied to reduce their susceptibility to side channels. Control-flow transformations, such as if-conversion, from their paper would aid in protecting sensitive functions from JIT-based side channels. In fact, there is existing work on compiler based strategies for mitigating side-channel vulnerabilities which might be germane to that purpose [55–57]. However, none of the solutions they offer have been integrated into the widely used runtimes explored in our work.

Frassetto et al. [58] propose JITGuard, a guard for JIT compilers against code-injection, code-reuse and data-only attacks. However, side-channel vulnerabilities are out of scope of their work. Ahead-of-time compilation, where a compilation to native machine code happens *before* execution rather than during, is a potential mitigation for JIT-induced side channels. Since compilation happens before runtime, it is agnostic to input distribution. Ahead-of-time

compilation is an option in some versions of Java [59, 60] and JavaScript [61, 62], but generally is not standard or even always available.

Side-Channel Analysis The problem of statically detecting side channels in software has been widely addressed, as discussed in the previous chapter. Static side-channel detection approaches rely on a cost model that statically approximates observable information (e.g., execution time) along a program path. What my work demonstrates is that such a cost model is insufficient. The execution time of a path depends not only on the instructions along that path but also, and to a great extent, on the state of the runtime. The state of the runtime is in turn influenced by all previous invocations of the code under test. Currently no static approach to side channel detection even attempts to model this complex interaction. Differential analyses are also used to automatically detect vulnerabilities in SSL/TLS [63]. Other dynamic approaches to side-channel detection analyze the network traffic of web applications in a black-box manner [30, 31, 64]. Type-based approaches to side-channel detection [27, 28] and approaches specific to cache-side channels [29, 65] have also been proposed. Currently, no static or dynamic approach to side-channel detection considers the state of the JVM as a factor in accessing side channels.

Other research is aimed towards the quantification of side-channel vulnerabilities [66, 67]. Research in this area aims at asking not simply if a program leaks information through a side channel, but *how much* information about the secret is leaked. These analyses rely on symbolic execution and suffer from poor scalability. Extensions to these approaches to perform attack synthesis on programs vulnerable to side channels [25, 68, 69] have also been explored.

Runtime-based CPU-induced side channels Branch prediction analysis (BPA) attacks and cache attacks are side-channel attacks which leverage runtime-dependent behavior of CPUs. Cache-based side-channel attacks [3, 70–74] have been theorized for years and have increasingly

been shown as a powerful technique for recovering sensitive information in practical scenarios. Acıiçmez et al. first demonstrated that the CPU's branch predictor could be leveraged to introduce timing channels in security-related code [75–77]. Since then, the CPU's Branch Prediction Unit has been exploited to introduce various flavors of timing channel vulnerabilities [78–80]. This work considers the impact of runtime behavior on the processor. We instead concern ourselves with its impact on the JVM.

3.9 Conclusions

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

Chapter 4

JVM Fuzzing for JIT-Induced Side-Channel Detection

In this chapter, I present an automated approach for finding JIT-induced side-channel vulnerabilities. I use this approach to demonstrate that programs labeled safe with respect to timing side channels by four different program analysis tools do in fact contain side-channel vulnerabilities if the runtime behavior of the program is sufficiently biased.

My contributions of this chapter are:

- An automated pattern-based approach for finding input partitions that are likely to be separable by a timing side channel.
- An automated technique to generate input belonging to different partition cells using branch instrumentation.
- An automated search strategy for a JVM state vulnerable to a timing channel using priming input generated via fuzzing.
- An evaluation of our approach on widely-used side-channel detection benchmarks, demonstrating its ability to automatically induce timing side channels in programs labelled safe

by four other recent analysis tools: Blazer [19], Themis [20], CoCo-Channel [81], and DiffFuzz [32].

The rest of this chapter is organized as follows. In Section 4.1, I discuss JIT-induced timing side channels and provide an overview of our automated detection process. In Section 4.2, I discuss my approach for pattern-based identification of partitions of secret input which are likely to be separable by a timing side channel. I detail how we instrument the program to generate test input data belonging to these partition cells. I also discuss how we use grey-box fuzzing to generate a set of priming inputs. In Section 4.3, I present an algorithm to use the generated test input data and priming input to fuzz the JVM for vulnerabilities to JIT-induced timing side channels. In Section 4.4, I evaluate our technique, discuss experiments and highlight key results. In Section 4.5, I present my conclusions.

4.1 Overview

Timing side-channel detection techniques investigate the following question: Assume a program p is a function on some input I consisting of secret input h and non-secret input l. For some fixed l, are there at least two mutually exclusive non-empty subsets of the secret domain h such that by observing the execution time of the program on a secret value, one can determine to which of the subsets the secret belongs? Note that precisely determining the answer to this membership question may not be possible due to either noise in the observations of the execution time or to overlapping execution time behavior of secret values belonging to different subsets of the input. So, typically, the answer we get to the question of "Does the secret belong to a particular subset?" is not a discrete "Yes" or "No" answer, but a quantitative answer that indicates our assessment of how likely it is that the secret belongs to that particular subset.

To automate the analysis we have to frame the problem in more detail. First, we need to characterize what type of information about the secret can be leaked from the program.

```
public static boolean sanity_unsafe(int a, int b) {
    int i = b;
    int j = b;
    if (b<0) return false;
    if (a<0) {
        return true;
    } else {
        while (i > 0) {
            i--;
        }
    }
        return false;
}
```

(b)

```
public static boolean sanity_safe(int a, int b) {
    int i = b;
    int j = b;
    if (b<0) return false;
    if (a>0) {
        while (i > 0) {
            i--;
        }
    } else {
        while (j > 0) {
            j--;
        }
    }
        return false;
}
```

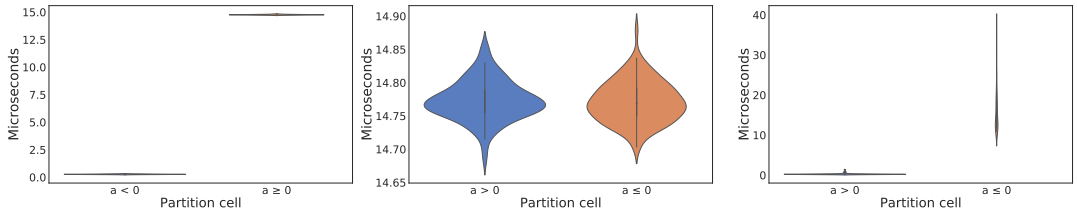
(a)

Figure 4.1: Two examples from the Blazer benchmark.

At a high level, the type of information that can be leaked about the secret will characterize the subsets of the secret domain for which the membership question is most likely answerable. One avenue through which information can be leaked is program branches. Each branch in the program can potentially influence the execution time. Therefore, the branches that depend on the secret can result in timing side channels.

Consider a secret-dependent branch and the corresponding branch condition, and assume that we divide the set of secret values to two subsets: 1) The secret values that cause the branch condition to evaluate to true, 2) The secret values that cause the branch condition to evaluate to false. Now, also assume that by observing the execution time, we can tell if the branch condition evaluates to true or not (this would be possible, for example, if the evaluation of the branch condition to true causes more expensive computation to be undertaken). In this case, we can conclude that there is a timing side channel and the program leaks information about the secret since we are able to distinguish which subset that the secret belongs to.

Let us make our discussion more concrete with an example. Figure 4.1(a) shows the sanity_unsafe method taken from the Blazer side-channel detection benchmark [19]. In this example the secret is the value of the argument a. The second if statement in the program corresponds to a branch that is dependent on the secret. If argument b is particularly large, it



(a) sanity_unsafe with JIT off (b) sanity_safe with JIT off (c) sanity_safe with JIT on

Figure 4.2: Execution time distributions for the sanity_safe and sanity_unsafe methods.

would be possible to detect if the then or else-branch is taken. This would enable us to tell if the secret value is less than zero. So, we can define two subsets of the secret domain (integer values less than zero, and integer values greater than or equal to zero), and by observing the execution time of the method we can tell which subset the secret belongs to. Hence, we can conclude that this method has a side-channel vulnerability and is unsafe. And, this is exactly what the side-channel detection tools report. Profiling the execution time of the sanity_unsafe method shows a clear timing side-channel. The violin plot in Figure 4.2(a) shows the execution time distributions for the sanity_unsafe method for partitions: $a < 0$ and $a \geq 0$ for a fixed b value.

Now, let us look at the sanity_safe method in Figure 4.1(b). Again, the secret is the value of the argument a, and the second if statement in the program corresponds to a branch that is dependent on the secret. However, for this case, both branches contain equivalent computations, so the same computation is performed regardless of the evaluation of the branch condition. This should imply that we cannot tell which branch is taken by observing the execution time.

All modern side-channel analysis tools [19, 20, 32, 81] perform this reasoning in one form or another and conclude that the method sanity_safe is indeed safe and does not contain a side-channel vulnerability. Let us again check this conclusion by profiling. Figure 4.2(b) demonstrates the execution time distribution for the sanity_safe method for partitions $a > 0$ and $a \leq 0$ with the same constant value for b as used in Figure 4.2(a). It is clear from these distributions

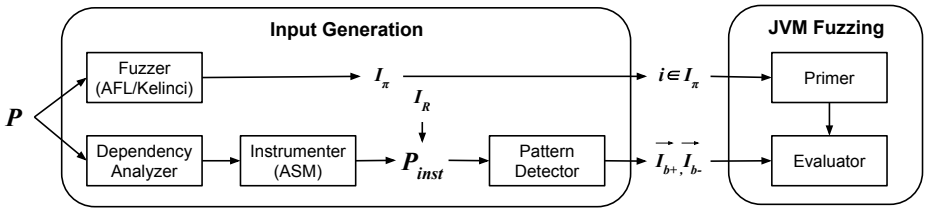


Figure 4.3: Automated JIT-Induced Side-Channel Detection

that an attacker cannot determine if the secret value is greater than zero or not by monitoring the execution time of the sanity_safe method since the timing behaviors for two cases are very similar, and, hence, there is no side-channel.

Unfortunately, this conclusion is wrong! The sanity_safe method shown in Figure 4.1(b) does contain a timing side-channel vulnerability. To understand why, we need to realize that the source code of sanity_safe is not the only factor on its execution time. The runtime environment itself plays a pivotal role in the program's behavior.

JIT-induced Side-Channels

The profiling data we show in Figure 4.2(b) was taken with JIT-disabled. If we enable JIT and then execute the sanity_safe method with a biased distribution where $a > 0$ is highly favored, JIT compilation will optimize this case. This leads to more efficient execution time for inputs where $a > 0$ and introduces a timing side channel leaking information about whether $a > 0$ or $a \leq 0$.

We profile again to determine if this hypothesis holds. Figure 4.2(c) shows the execution time distribution for the sanity_safe method again for the partitions: $a > 0$ and $a \leq 0$. We obtained these distributions by enabling JIT and repeatedly executing sanity_safe with an input distribution heavily favoring values where $a > 0$. After this priming stage, we timed calls to sanity_safe on input values from different partition cells and created a violin plot of the resulting

timing distributions. We again kept the value of the public input b constant. The timing side channel is apparent from the clearly separable distributions for the two partition cells.

Currently, all side-channel analysis detection tools miss this vulnerability. This is due to its inception in the runtime behavior and in the biased input distribution provoking the JVM into favoring some program paths. Currently, no side-channel analysis tool can detect this class of side-channel vulnerabilities. A bias in the input distribution can arise for a variety of reasons. If an attacker can call a method repeatedly, then the attacker can force the JIT compiler to optimize a particular execution path going through a secret dependent branch. There might also be a natural bias in the input distribution. For example, if it is the case that most users call the sanity_safe method with a secret value $a > 0$ (or visa versa), then a JIT-induced side channel can arise based on this common behavior.

Ultimately, the key factor in forcing JIT compilation to induce a side channel is bias in the input distribution. This is a dynamic characteristic and the same program can be fed inputs in a countless variety of ways. I use the term *priming* to mean the act of executing a program repeatedly with a certain distribution of program input. Three factors determine a priming strategy: the program input used, the number of program calls made, and the bias in the input distribution. This huge search space of possible priming strategies calls for a systematic strategy of exploration if we have any hope of automatically detecting a program's vulnerability to JIT-induced side channels.

Automating JIT-Induced Side Channel Detection

Our automated approach consists of two main phases, outlined in Figure 4.3. The first phase is the Input Generation phase. This phase has two end goals. Given a program p taking secret input, we first will generate pairs of partition cells (I_{b^+}, I_{b^-}), where each cell is a subset of the secret domain. These pairs of partition cells are candidates of mutually exclusive non-empty subsets of the secret domain that may be distinguishable via a JIT-induced side channel.

In order to generate these partition cells, we define six program behavior patterns to identify input partitions potentially prone to side-channel vulnerabilities. We use static dependency analysis to identify secret dependent branches to use in these patterns. Then, we create an instrumented version of the program p_{inst}, introducing two counters for each branch. The first counts how many times the branch condition is reached. The second counts how many times branch condition evaluates to true. We generate a set of inputs I_R, run p_{inst} on these inputs and collect information for each input based on the branch counter values. We match these values to pre-defined behavior patterns which partition I_R to yield the desired (I_{b^+}, I_{b^-}) pairs. The lower half of the Input Generation block of Figure 4.3 overviews this generation of (I_{b^+}, I_{b^-}) evaluation pairs.

The second goal of the Input Generation phase is to generate a set of priming inputs I_π. Values from $I_\pi$are used to prime the JVM to favor certain program paths in an attempt to introduce timing side channels. We use the grey-box fuzzer AFL [82] to generate inputs executing different paths through the program. Input values that cover different program paths enable us to search different ways of priming the JVM while searching for a JVM state vulnerable to timing side channels.

The completion of the Input Generation phase yields a set of priming inputs I_π and a set of pairs of test partition cells $(\vec{I}_{b^+}, \vec{I}_{b^-})$. The following JVM Fuzzing phase answers the question: For any given pair $(I_{b^+}, I_{b^-}) \in (\vec{I}_{b^+}, \vec{I}_{b^-})$ of input partition cells, does priming the JVM in favor of some input $i_\pi \in I_\pi$result in a timing side channel through which input from $I_{b^+}$and $I_{b^-}$are distinguishable by their execution time? To answer, we systematically explore different priming strategies by iterating over different priming amounts and priming ratios. We evaluate if we successfully induced a timing side channel by quantifying the information leakage.

4.2 Input Generation

Below I discuss generation of input partitions and priming inputs.

4.2.1 Secret-Dependent Branch Detection

The first step of our approach is to identify secret dependent branches. A branch is secret dependent if the evaluation of the branch condition depends on the value of a secret program input. Given a program and a set of inputs marked as secret, we use static dependency analysis to identify the secret dependent branches.

We used JANALYZER [83], an existing static analysis tool, to perform this dependency analysis. JANALYZER constructs the system dependence graph (SDG) [84] for a given program. A SDG is constructed by first extracting the call graph for the program and the procedure dependence graph (PDG) [85] for each procedure. Nodes of the PDG are either statements or branch conditions and edges represent dependencies. A data dependence is a triple (d, u, v) calculated using reaching definitions analysis [86] where v is a variable and d and u are PDG nodes, v is defined in node d, used in node u, there is a path from d to u, and v is not redefined in between. For example, for the sanity_unsafe procedure shown in Figure 4.1(a), one such data dependence triple is (*argument a*, *if (a<0)*, *a*). Using SDG and PDGs, we do forward dependency analysis from all the marked secret program inputs and identify the program statements and branch conditions dependent on those inputs. We filter out all other statements keeping only the branch conditions as we are concerned about secret dependent branches only.

In our experiments, we observed that using both data and control dependencies introduces too many spurious dependencies. Hence, we consider only data dependencies during PDG construction but track both explicit (direct data flow using reaching definitions analysis) and implicit flows (data flows to v occurs inside a branch whose predicate is dependent on secret) following [20]. The end result is a set of secret-dependent branches.

Removing control dependencies leads to unsoundness in the dependency analysis, and our side-channel detection technique can generate false negatives, in the sense that if we are unable to detect a JIT-induced side-channel, that does not guarantee that the program is free of JIT-induced side channels. On the other hand, our side-channel detection technique does not generate false positives, i.e., when we do report a JIT-induced side channel, it means that there is demonstrable side channel in the program.

Although ignoring control dependencies reduces the number of spurious dependencies, data flow analysis can still lead to spurious dependencies. However, since we actually execute the program on input values we generate, these will not lead to any false reports of JIT-induced side channels.

4.2.2 Pattern Detection

A timing side channel is present if there are at least two mutually exclusive subsets of the secret domain such that an attacker who is able to observe the execution time of the program can determine which of the two subsets an unknown secret value belongs to (under the constraint that the secret does belong to one of the two). It is not feasible to explore the entire set of possible partitions of the secret domain to look for such subsets, and randomly chosen partitions are unlikely to yield meaningful results. To reduce the space of potential partition choices, we observe that timing side channels typically arise due to secret-dependent branches whose evaluation leads to an observable difference in the execution time.

We identify six program behavior patterns with respect to branch conditions corresponding to if statements, and use these patterns to define partitions of secret input. Matches to these patterns can be detected via instrumentation as we describe later in this section. One can define more patterns and corresponding partitions, however, our experiments demonstrate that these six are rich enough to effectively detect JIT-induced side-channel vulnerabilities.

Let I denote the set of all inputs for a given program p, where, given an input $i \in I$, $p(i)$ denotes the run that corresponds to executing program p on input i. We use h to denote the secret part of the input and l to denote the non-secret (public) part. Given an input $i \in I$ to the program, we write $i = \langle l, h \rangle$ to denote that the input consists of concatenation of public and secret parts. Given a program p, let b_p denote the set of branches in the program and $b_h \subseteq b_p$ denote the set of branches with branch conditions that are dependent on the secret h.

For any run $p(i)$ of the program and any $b \in b_p$, let $\#(p(i), b)$ be the total number of times the branch b is reached during the run $p(i)$ (i.e., the number of times the condition guarding b is evaluated during the run $p(i)$). Let $\#(p(i), b^+)$ denote the number of times the branch condition for b evaluates to true and $\#(p(i), b^-)$ denote the number of times the branch condition for b evaluates to false. Note that, for all p, b, and i, $\#(p(i), b) = \#(p(i), b^+) + \#(p(i), b^-)$.

Given a set of inputs I, our goal is to partition I using a secret-dependent branch $b \in b_h$. Given a branch $b \in b_h$, we partition I into I_{b^+}, I_{b^-} and I_0 where $I = I_{b^+} \cup I_{b^-} \cup I_0$ and I_{b^+}, I_{b^-} and I_0 are mutually exclusive. Below, we define six patterns we use for obtaining I_{b^+}, I_{b^-} and I_0.

We first consider the following pattern: $b \in b_h$ corresponds to an if statement which is not in a loop or a recursive function and can be reached at most once per each run of the program. We call this *Single Visit (SV)* branch pattern. For the SV pattern, we define I_{b^+}, I_{b^-} and I_0 as follows:

1. $\forall i \in I_{b^+} \cup I_{b^-}$, $\#(p(i), b) = 1$.
2. $\forall i \in I_{b^+}, \#(p(i), b^+) = 1$.
3. $\forall i \in I_{b^-}, \#(p(i), b^-) = 1$.
4. $I_{b^+} \neq \emptyset \wedge I_{b^-} \neq \emptyset$.
5. $\forall i_1, i_2 \in I_{b^+} \cup I_{b^-}, \forall b_1 \in b_h \cup b_l, b_1 \neq b$,
   $\#(p(i_1), b_1) = \#(p(i_2), b_1) \wedge \#(p(i_1), b_1^+) = \#(p(i_2), b_1^+)$.

Intuitively, this partition divides the set of inputs I into those that generate runs in which b is reached and the branch condition evaluates to true (I_{b^+}, (2)) and those that generate runs in which b is reached and branch condition evaluates to false (I_{b^-}, (3)) subject to the condition that all these runs agree on all other branch condition evaluations (5). We also require I_{b^+} and I_{b^-} to be non-empty (4). All inputs that cannot be put into I_{b^+} or I_{b^-} are put into I_0.

As an example, for the sanity_unsafe procedure shown in Figure 4.1(a), using the SV pattern we obtain the following partition of the input domain: $I_{b^+} = \{a \mid a < 0\}$, $I_{b^-} = \{a \mid a \geq 0\}$ and $I_0 = \emptyset$,

I_{b^+} and I_{b^-} define two subsets of the secret domain whose separability via timing observations we wish to evaluate. Note that multiple partitions can satisfy the above criteria depending on the behavior of the runs that are generated by the inputs in I_{b^+} and I_{b^-} regarding other conditional branches. We can explore different partitions and if we can find one partition in which membership to I_{b^+} or I_{b^-} can be determined by timing observations, then we can conclude that there is a timing side-channel.

The requirement that runs generated by inputs in I_{b^+} and I_{b^-} must agree on all branch condition evaluations barring those on b itself is strong. It is possible that there might be no such partition cells I_{b^+} and I_{b^-}. For example, the evaluation of b to true could generate runs that reach a branch condition unreachable should b evaluate to false. This motivates our *Single Visit Relaxed (SVR)* branch pattern. Under this pattern, I_{b^+} and I_{b^-} are defined using the same rules for the SV pattern, except the rule (5) is modified as:

$\forall i_1, i_2 \in I_{b^+} \cup I_{b^-}, \forall b_1 \in b_h \cup b_l, b_1 \neq b,$

$(\#(p(i_1), b_1) > 0 \wedge \#(p(i_2), b_1) > 0) \Rightarrow$

$(\#(p(i_1), b_1) = \#(p(i_2), b_1) \wedge \#(p(i_1), b_1^+) = \#(p(i_2), b_1^+)).$

Now, runs generated from I_{b^+} and I_{b^-} need only agree on all branch evaluations reached by every run. Through another lens, this means that all runs share a common, possibly empty prefix and common, possibly empty suffix. In practice, we look for partitions that maximize the

size of this shared prefix and suffix.

Next, let us consider a branch $b \in b_h$ corresponding to an if statement where b is either in a loop or in a recursive function. Hence, b might be reached multiple times. Again, there are many ways to partition the input domain based on such a branch condition. We will focus on four patterns: 1) *Multiple Visit All True (MVAT)* and 2) *Multiple Visit All False (MVAF)* and the relaxed variants of these two patterns 3) *Multiple Visit All True Relaxed (MVATR)* and 4) *Multiple Visit All False Relaxed (MVAFR)* which we define below.

In the MVAT pattern the branch b is visited multiple times. I_{b^+} consists of input values that generate runs where every time b is reached, the branch condition evaluates to true. I_{b^-} consists of input values that generate runs where at least one time when b is reached, the branch condition evaluates to false. We formalize the MVAT pattern as follows:

1. $\forall i \in I_{b^+} \cup I_{b^-}, \#(p(i), b) > 1$.
2. $\forall i \in I_{b^+}, \#(p(i), b^+) = \#(p(i), b)$.
3. $\forall i \in I_{b^-}, \#(p(i), b^-) \geq 1$.
4. $\forall i_1 \in I_{b^+}, \forall i_2 \in I_{b^-}, \#(p(i_1), b) = \#(p(i_2), b)$.
5. $I_{b^+} \neq \emptyset \wedge I_{b^-} \neq \emptyset$.
6. $\forall i_1, i_2 \in I_{b^+} \cup I_{b^-}, \forall b_1 \in b_h \cup b_l, b_1 \neq b,$
   $\#(p(i_1), b_1) = \#(p(i_2), b_1) \wedge \#(p(i_1), b_1^+) = \#(p(i_2), b_1^+)$.

As with the SV pattern, we require that I_{b^+} and I_{b^-} are non-empty (5). We also require that all runs generated from inputs in I_{b^+} or I_{b^-} reach the branch b same number of times and agree on all branch condition evaluations other than the ones for the branch b (6). All inputs that cannot be put into I_{b^+} or I_{b^-} due to above constraints are put into I_0.

In the MVAF pattern the branch b is visited multiple times. I_{b^+} consists of input values that generate runs where at least one time when b is reached, the branch condition evaluates to

true. I_{b^-} consists of input values that generate runs where every time b is reached, the branch condition evaluates to false. We formalize the MVAF pattern as:

1. $\forall i \in I_{b^+} \cup I_{b^-}$, $\#(p(i), b) > 1$.
2. $\forall i \in I_{b^+}, \#(p(i), b^+) \geq 1$.
3. $\forall i \in I_{b^-}, \#(p(i), b^-) = \#(p(i), b)$.
4. $\forall i_1 \in I_{b^+}, \forall i_2 \in I_{b^-}, \#(p(i_1), b) = \#(p(i_2), b)$.
5. $I_{b^+} \neq \emptyset \wedge I_{b^-} \neq \emptyset$.
6. $\forall i_1, i_2 \in I_{b^+} \cup I_{b^-}, \forall b_1 \in b_h \cup b_l, b_1 \neq b$,
   $\#(p(i_1), b_1) = \#(p(i_2), b_1) \wedge \#(p(i_1), b_1^+) = \#(p(i_2), b_1^+)$.

We also define two more patterns by relaxing the requirements for the MVAT and MVAF patterns. For the MVATR pattern, I_{b^+} and I_{b^-} are defined as:

1. $\forall i \in I_{b^+} \cup I_{b^-}$, $\#(p(i), b) \geq 0$.
2. $\forall i \in I_{b^+}, \#(p(i), b^+) = \#(p(i), b)$.
3. $\forall i \in I_{b^-}, \#(p(i), b^-) \geq 1$.
4. $I_{b^+} \neq \emptyset \wedge I_{b^-} \neq \emptyset$.
5. $\forall i_1, i_2 \in I_{b^+} \cup I_{b^-}, \forall b_1 \in b_h \cup b_l, b_1 \neq b$,
   $\#(p(i_1), b_1) = \#(p(i_2), b_1) \Rightarrow \#(p(i_1), b_1^+) = \#(p(i_2), b_1^+)$.

where the requirement that branch condition b is reached the same number of times across runs generated by input in I_{b^+} or I_{b^-} is relaxed. This extends the applicability of the pattern to programs where the choice on condition b might cause the loop or recursive function containing b to terminate.

Similarly, we define the MVAFR pattern as:

1. $\forall i \in I_{b^+} \cup I_{b^-}$, $\#(p(i), b) \geq 0$.

2. $\forall i \in I_{b^+}, \#(p(i), b^+) \geq 1$.

3. $\forall i \in I_{b^-}, \#(p(i), b^-) = \#(p(i), b)$.

4. $I_{b^+} \neq \emptyset \wedge I_{b^-} \neq \emptyset$.

5. $\forall i_1, i_2 \in I_{b^+} \cup I_{b^-}, \forall b_1 \in b_h \cup b_l, b_1 \neq b$,
   $\#(p(i_1), b_1) = \#(p(i_2), b_1) \Rightarrow \#(p(i_1), b_1^+) = \#(p(i_2), b_1^+)$.

In the MVATR and MVAFR patterns, similar to the the SVR pattern, the requirement that all runs generated from input in I_{b^+} or I_{b^-} agree on all branch evaluations bar b itself is relaxed. We only require agreement on all branch conditions reached the same number of times across the runs. Once again, in practice we choose partitions that maximize the number of conditions agreed upon.

4.2.3 Branch Instrumentation

Given a program p we use dependency analysis discussed earlier to determine the set of secret dependent branches b_h. Then we instrument the program in order to generate sets of inputs based on the patterns above.

Given the program p and the secret dependent branches b_h we use Java bytecode manipulation and analysis framework ASM [87] to generate an instrumented program p_{inst} by introducing two counters (i.e., integer variables) for each secret-dependent branch b. One counter c_b is initialized to 0 and is incremented by one every time program execution reaches b. The other counter c_{b^+} is initialized to 0 and is incremented by one every time the program execution reaches b and the branch condition for b evaluates to true. The instrumented program p_{inst} prints the values of these counters when the program terminates. Given an input i, let $p_{inst}(i)[c_b]$ and $p_{inst}(i)[c_{b^+}]$ denote the values of c_b and c_{b^+} printed by p_{inst} when it is run using input i. Then, we have the following equivalences:

$$\#(p(i), b) = p_{inst}(i)[c_b], \qquad \#(p(i), b^+) = p_{inst}(i)[c_{b^+}],$$
$$\#(p(i), b^-) = p_{inst}(i)[c_b] - p_{inst}(i)[c_{b^+}].$$

Next, we generate a set of inputs $I_R \subseteq I$ and run p_{inst} on each $i \in I_R$. We focus on programs where the secret input is either string or numeric, and we use built-in Java functions returning pseudorandom values to build I_R. Then, we look at the output of each p_{inst} and using the patterns SV, MVAT, and MVAF and their relaxed variants we identify subsets $I_{b^+} \subseteq I_R$ and $I_{b^-} \subseteq I_R$ for each secret dependent branch b if the outputs generated by p_{inst} on I_R match these patterns. The result of this step is pairs of subsets $I_{b_1^+}, I_{b_1^-}$; $I_{b_2^+}, I_{b_2^-}$; ..., one per branch condition. We denote them as vectors $\vec{I}_{b^+}$, $\vec{I}_{b^-}$, and use them to search for JIT-induced side channels during the JVM Fuzzing phase.

4.2.4 Input Generation for Priming

JIT-induced side channels arise when a bias in the input distribution to a program causes a program path to "heat up," speeding up its execution relative to other program paths. To automate the detection of possible side channels arising from JIT, we generate a set of possible priming inputs $I_\pi \subseteq I$. The ideal set of priming inputs are values that exercise different program paths. During JVM Fuzzing phase we evaluate each priming input $i_\pi \in I_\pi$ to determine its effectiveness at inducing a timing side channel.

We use Kelinci [88], an interface to run the grey-box fuzzer American Fuzzy Lop (AFL) [82] on Java programs, to generate I_π. AFL is a genetic fuzzer designed to automatically discover interesting test cases that trigger new behaviors in the targeted program. AFL has been widely and successfully used, finding hundreds of high-impact vulnerabilities [89] and has a large community of active users. AFL is a coverage-based fuzzer, employing a variety of strategies and heuristics to generate inputs that traverse different paths in the program. It uses several different techniques such as bit-flipping and sequential insertion of known interesting integers to

effectively trigger new program behaviors. AFL stores a witness $i \in I$ for each new program state found. We use the complete set of witnesses found upon termination of fuzzing as I_π. Though there is no guarantee that AFL will be able to generate an input for every program path, it has low overhead and less limitations in comparison to heavy-weight analyses such as symbolic execution, making it a better choice for our automated analysis.

4.3 JVM Fuzzing for Side Channels

Our goal is to determine if there is any priming input $i_\pi \in I_\pi$ such that priming in favor of i_π results in a timing side channel allowing an attacker to distinguish I_{b^+} from I_{b^-} for any two parition cells of the input domain as defined in Section 4.2.

Priming the JVM

We provide the pseudocode for our priming procedure in Algorithm 5. Given a certain priming amount n, a distribution α, a priming value i_π, and a set of other input values I_π-i_π, the program p is called n times in total. Of those n times, an α fraction of the calls are on the input i_π. The rest of the calls are on a randomly chosen input value drawn from I_π-i_π. Intuitively, the ratio α exists to model the case where the attacker does not have complete control over the JVM and therefore cannot prime perfectly in favor of i_π. The lower the α ratio is, the less control we assume the attacker has. We experiment with varied α values to explore what percentage of calls to p an attacker needs to control in order to induce a timing side channel. Similarly, we vary the value of n to explore the needed amount of priming. Both of these parameters inform under what scenarios an attacker will be able to induce a timing side channel in program p.

Evaluating JVM Vulnerability

For a given priming of the JVM, we wish to evaluate if a timing side channel has been introduced that leaks information about the membership of some secret input i to sets I_{b^+} and I_{b^-} based on the timing of $p(i)$. Algorithm 6 outlines this evaluation process. Given the subsets I_{b^+} and I_{b^-}, we first prime the JVM as described above and then time a call to p on a random input i drawn from I_{b^+}. We collect a timing distribution for I_{b^+} by repeating this process N times. Various sources of non-determinism, from system noise to variations in runtime decisions made by the JIT compiler, affect the time measurements. The higher the value of N, the more robust the statistical profile is to such noise. We repeat this process to also collect a timing distribution for I_{b^-}.

Given the timing distributions for I_{b^+} and I_{b^-}, we compute the conditional entropy between the membership of i and the timing of $p(i)$ after priming the JVM. From this, we report how much information about the membership of i we can expect to be leaked from a single time measurement. The membership of i encodes one bit of information. Therefore a value of 1 means full leakage of the membership of i and a value of 0 means no leakage.

Parameter Exploration for JVM Fuzzing

For each program p, we have a set of pairs of input cells $(\vec{I}_{b^+}, \vec{I}_{b^-})$ generated as described in Section 4.2. We also have a set of priming inputs I_π generated for the same program using Kelinci and AFL as described in Section 4.2. Now, we iterate over these parameters, in essence fuzzing the JVM in an attempt to find a JVM state vulnerable to a timing side channel. This process is described in Algorithm 7. For each $i_\pi \in I_\pi$ and pair of input sets $(I_{b^+}, I_{b^-}) \in (\vec{I}_{b^+}, \vec{I}_{b^-})$, we evaluate if priming in favor of i_π induces a side channel allowing us to learn the membership of an input i in sets I_{b^+} and I_{b^-}. The runtime decisions controlling JIT compilation are complex, meaning that the number of priming iterations and priming distribution can influence whether

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

Algorithm 5: Priming

```
input : N (profiling amount), n (priming amount), α (ratio),
        i_more, I_less(priming inputs), I_b+, I_b−(profiling test input sets)
v_b+, v_b− ← two empty vectors to store timing profiles;
for i ← 1 to N do
    Prime(n, α, i_π, I_π- i_π );
    v_b+.append( Time(p(i_b+ ← random(I_b+)) // start with fresh JVM
end
for i ← 1 to N do
    Prime(n, α, i_π, I_π- i_π );
    v_b−.append( Time(p(i_b− ← random(I_b−)) // start with fresh JVM
end
ComputeConditionalEntropy(v_b+, v_b−);
```

Algorithm 6: Evaluation

a timing channel is induced. Therefore, we also explore those parameters, iterating over a set of potential priming amounts and distributions.

4.4 Experimental Evaluation

Datasets

The Blazer dataset consists of 24 benchmarks drawn from a combination of challenge programs from the DARPA STAC program, classic examples from the literature [21–23], and microbenchmarks constructed by the Blazer authors. The 24 benchmarks consist of 12 unsafe programs and their "safe" variants. Table 4.6a shows the LOC for the safe Blazer variants (unsafe variants are similar. Four different program analysis tools for automated side-channel

```
input : N (profiling amount), 𝒩 (priming amounts), A (ratios),
        I_π(priming inputs), (I⃗_{b+}, I⃗_{b−}) (profiling test input sets)
for (I_{b+}, I_{b−}) ∈ (I⃗_{b+}, I⃗_{b−}) do
    for i_π ∈ I_π do
        for n ∈ 𝒩 do
            for α ∈ A do
                VulnerabilityEvaluation(N, n, i_π, I_π- i_π, I_{b+}, I_{b−})
            end
        end
    end
end
```

Algorithm 7: JVM Fuzzing

detection have used this benchmark for evaluation. Blazer [19], Themis [20], and CoCo-Channel [81] are state-of-the-art static analysis tools for detecting timing side channels in Java bytecode. They each report the vulnerability of the unsafe program variants and the safety of the patched versions. DiffFuzz [32] is a dynamic analysis tool for the detection of side channels. DiffFuzz catches an overflow bug resulting in a timing side channel in loopAndbranch_safe and considers that the safety of gpt14_safe is questionable depending on an observability threshold. Otherwise, DiffFuzz agrees with the verdicts of the other tools.

The Themis dataset consists of 10 unsafe programs and their "safe" variants. We evaluate on the 8 unsafe programs that contain timing side channels and their "safe" variants. Table 4.6b shows the LOC for the safe Themis variants. Outside of a benchmark from JDK6, the rest are Java programs collected from Github. Themis, CoCo-Channel and DiffFuzz have all used this benchmark for evaluation, agreeing on the safety of all safe variants, bar jetty_safe where DiffFuzz detects a subtle side channel.

The DiffFuzz dataset consists of 5 unsafe programs with timing side channels and their safe variants used in addition to the Blazer and Themis benchmarks to evaluate DiffFuzz [32]. Table 4.6c shows the LOC for the safe DiffFuzz variants. Four of these programs and their corresponding safe variants are drawn from the open source project Apache FtpServer [90] and the last is from an open source authentication plugin for Minecraft servers available on Github [91]. We exclude ibasys from the DiffFuzz dataset since its secret input is an image.

Table 4.1: Experimental results for best leakage in the Blazer (top), Themis (middle) and DiffFuzz (bottom) safe variants.

Program variant	**Inst. pattern**	**Best leakage**					
		leakage	n	α	avg. time (ns)		
					I_{b+}	I_{b-}	diff
`array_safe`	SV	0.995	100000	0.9990	205	16335	16130
`loopBranch_safe`	SV	1.000	100000	0.9990	192	16020	15828
`sanity_safe`	SV	0.995	100000	0.9990	199	13382	13183
`straightline_safe`	SV	0.995	100000	0.9980	200	15927	15727
`unixlogin_safe`	MVAF	0.995	10000	0.9980	329	17553	17224
`modPow1_safe`	MVAT	0.977	1000	0.9000	16977	8512	8465
`modPow2_safe`	MVAT	0.995	1000	0.9990	34423	7853	26570
`pwdEqual_safe`	MVAF	0.485	500000	0.9999	547	12870	12323
`pwdEqual_safe`	MVAT	0.744	500000	0.9999	550	15877	15337
`pwdEqual_safe`	SV	0.849	500000	0.9990	18642	529	18113
`k96_safe`	MVAT	0.995	1000	0.9000	19952	8547	11405
`gpt14_safe`	MVAT	0.862	10000	0.9980	15434	43265	27831
`login_safe`	SV	0.967	500000	0.9999	30769	3191	27578
`jetty_safe`	SV	1.000	100000	0.9980	302	16617	16315
`jetty_safe`	MVAF	0.873	100000	0.9980	891	16108	15217
`jetty_safe`	MVAT	0.860	100000	0.9990	271	16842	16571
`spring_safe`	SV	1.000	100000	0.9980	719	19123	18404
`tomcat_safe`	SVR	1.000	100000	0.9990	8943	261209	252266
`pac4j_safe`	SVR	1.000	10000	0.9980	19223	188121	168898
`apache_clear_safe`	SV	0.970	10000	0.9990	4677	50015	45338
`apache_stringUtils_safe`	MVAF	0.612	1000	0.9500	1612	2014	402

Such complex data structures can be handled by our approach by providing pseudorandom input generators for such domains. However, as previously stated, our current implementation focuses on numeric and string secret domains.

Hardware Setup. All experiments were run on a computer equipped with an Intel i5-6600K CPU at 3.50 GHz and 32 GB of RAM running Ubuntu Linux 16.04 (Linux 4.4.0-103) and the Java 8 Platform Standard Edition, version 1.8.0_162, from OpenJDK.

Experimental Setup. Each program variant comes with a set of inputs marked as secret. For each safe program variant, we first conducted dependency analysis as described in Section 4.2.1, yielding a set of secret-dependent branches. Using these branches, we generated the instrumented programs per Section 4.2.3. We then proceeded to generate random secret input to the instrumented program. We used our pattern detection scheme described in Section 4.2.2 on the results to generate sets of possible partitions cells. Focusing on programs with numeric or string input values enabled us to write a pseudorandom input generator for each variant. For the

Table 4.2: Experimental results for limited leakage for Blazer (top), Themis (middle) and DiffFuzz (bottom) safe variants.

Program variant	**Inst. pattern**	**Limited leakage**					
		leakage	n	α	avg. time (ns)		
					I_{b+}	I_{b-}	diff
`array_safe`	SV	0.741	1000	0.9000	359	434	75
`loopBranch_safe`	SV	0.953	1000	0.9000	476	338	138
`sanity_safe`	SV	0.337	1000	0.9000	725	685	40
`straightline_safe`	SV	0.810	1000	0.9000	397	477	80
`unixlogin_safe`	MVAF	0.519	1000	0.9000	620	639	19
`modPow1_safe`	MVAT	0.977	1000	0.9000	16977	8512	8465
`modPow2_safe`	MVAT	0.971	1000	0.9000	24520	8809	15711
`pwdEqual_safe`	MVAF	0.357	10000	0.7000	378	344	34
`pwdEqual_safe`	MVAT	0.597	10000	0.9000	400	294	106
`pwdEqual_safe`	SV	0.635	1000	0.9000	723	930	207
`k96_safe`	MVAT	0.995	1000	0.9000	19952	8547	11405
`gpt14_safe`	MVAT	0.306	10000	0.9000	28290	21738	6552
`login_safe`	SV	0.571	10000	0.7000	432	556	124
`jetty_safe`	SV	0.510	1000	0.9000	610	752	142
`jetty_safe`	MVAF	0.253	1000	0.9000	624	687	63
`jetty_safe`	MVAT	0.178	10000	0.9000	291	362	71
`spring_safe`	SV	0.067	10000	0.7000	697	774	77
`tomcat_safe`	SVR	1.000	1000	0.9000	16801	184082	167281
`pac4j_safe`	SVR	1.000	10000	0.7000	20076	95022	74946
`apache_clear_safe`	SV	0.026	1000	0.7000	5695	5721	26
`apache_stringUtils_safe`	MVAF	0.307	10000	0.9000	1116	1375	259

modular exponentiation cases and password comparison functions, we placed an upper bound on the bit length of the input integer or length of the input string. We fixed any public input value arbitrarily. We began matching with the more strict patterns (SV, MVAT, MVAF), but if no partitions were found, we proceeded to their relaxed variants. We stopped the process for each applicable pattern as soon as a partition was found. The end result is a vector of sets ($\vec{I}_{b+}$, $\vec{I}_{b-}$).

We wrote a driver for Kelinci to generate the set of priming input values I_π for each safe variant. As AFL requires a directory of well-formed seed input, we generated one or two seed inputs per variant. We allowed Kelinci to run on each variant for at most one hour and used the set of witnesses for different program states found as I_π for the variant.

With the Input Generation Phase complete, we performed the JVM fuzzing given in Algorithm 7. We used a set of priming numbers, $n = \{1000, 10000, 100000, 500000\}$ and distributions, $\alpha = \{0.5, 0.7, 0.9, 0.95, 0.998, 0.999, 0.9999\}$ to determine the vulnerability of the resulting JVM

Table 4.3: Experimental results with JIT disabled for Blazer (top), Themis (middle) and DiffFuzz (bottom) safe variants.

Program variant	**Inst. pattern**	**Leakage w/o JIT**			
		leakage	avg. time (ns)		
			I_{b+}	I_{b-}	diff
array_safe	SV	0.029	286	287	1
loopBranch_safe	SV	0.206	267	287	20
sanity_safe	SV	0.026	343	350	7
straightline_safe	SV	0.025	373	377	4
unixlogin_safe	MVAF	0.114	604	640	36
modPow1_safe	MVAT	0.033	37095	37342	247
modPow2_safe	MVAT	0.397	36491	38220	1729
pwdEqual_safe	MVAF	0.077	857	822	35
pwdEqual_safe	MVAT	0.363	782	881	99
pwdEqual_safe	SV	0.051	877	902	25
k96_safe	MVAT	0.078	37065	37276	211
gpt14_safe	MVAT	0.071	37305	37654	349
login_safe	SV	0.354	3683902128	3686425340	2523212
jetty_safe	SV	0.120	720	759	39
jetty_safe	MVAF	0.026	722	726	4
jetty_safe	MVAT	0.074	721	751	30
spring_safe	SV	0.015	4111	4198	87
tomcat_safe	SVR	1.000	27907	168234	140327
pac4j_safe	SVR	0.980	270478	335170	64692
apache_clear_safe	SV	0.020	195067	196100	1033
apache_stringUtils_safe	MVAF	0.352	5391	5709	318

primed in favor of each $i_\pi \in I_\pi$. We used a profiling amount of $N = 100$. As a baseline, we ran each safe variant with JIT disabled and gathered timing information for each $(\vec{I}_{b+}, \vec{I}_{b-})$ pair. Since priming the JVM does not matter in this case, we did not explore the parameter space for these runs.

To determine the strength of the resulting timing channel, we calculate the entropy of partition cell membership conditioned on timing observation. The leakage reported is $1-$ this conditional entropy. We profile multiple times to gain probability distributions for observed execution times.

As noted by the authors of DiffFuzz [32], a NullPointerException in unixlogin_safe prevents it from being executable. We fixed the issue by replacing the culprit hash comparison with a dummy comparison. As discussed by the authors of CoCo-Channel, some secret-dependent conditionals (such as certain null pointer checks) in the Themis dataset were manually marked by the Themis authors as secret-independent. We ignored the same set of

Table 4.4: Experimental results for best leakage for Blazer (top), Themis (middle) and DiffFuzz (bottom) unsafe variants.

Program variant	**Inst. pattern**	**Best leakage**					
		leakage	n	α	avg. time (ns)		
					I_{b+}	I_{b-}	diff
array_unsafe	SV	0.995	1000	0.9000	345	234	111
loopBranch_unsafe	SV	0.924	100000	0.9990	16325	197	16128
notaint_unsafe	SV	1.000	1000	0.5000	1184025	306	1183719
sanity_unsafe	SV	0.995	100000	0.9980	13757	204	13553
straightline_unsafe	SV	0.995	100000	0.9990	16136	196	15940
unixlogin_unsafe	MVAF	0.459	1000	0.7000	544	598	54
modPow1_unsafe	MVAT	0.287	100000	0.9500	5661	4716	945
modPow2_unsafe	MVAT	0.256	1000	0.9000	2329	2802	473
pwdEqual_unsafe	MVAF	0.353	100000	0.5000	614	574	40
pwdEqual_unsafe	MVAT	0.813	1000	0.9000	809	652	157
pwdEqual_unsafe	SV	0.664	1000	0.9980	691	573	118
k96_unsafe	MVAT	0.821	10000	0.9990	76023	28853	47170
gpt14_unsafe	MVAT	0.995	100000	0.9990	1554	35114	33560
login_unsafe	SV	0.937	500000	0.9999	5116	29735	24619
bootauth_unsafe	MVATR	0.980	100000	0.9980	3006	30653	27647
bootauth_unsafe	MVAFR	1.000	100000	0.9980	3049	39056	36007
jdk_unsafe	MVATR	0.710	100000	0.9980	1073	18290	17217
jdk_unsafe	MVAFR	0.400	100000	0.9980	2272	14573	12301
jetty_unsafe	SV	1.000	10000	0.9000	307	749	442
jetty_unsafe	MVAFR	0.554	100000	0.9990	369	15151	14782
jetty_unsafe	MVATR	0.780	1000	0.9990	510	1019	509
orientdb_unsafe	MVATR	1.000	500000	0.9999	436	9818	9382
orientdb_unsafe	MVAFR	0.244	1000	0.9500	623	735	112
picketbox_unsafe	MVATR	0.755	100000	0.9990	795	19197	18402
picketbox_unsafe	MVAFR	0.417	500000	0.9999	1188	6925	5737
spring_unsafe	SV	1.000	100000	0.9990	656	19705	19049
tomcat_unsafe	SVR	1.000	10000	0.7000	6699	246234	239535
pac4j_unsafe	SVR	1.000	10000	0.9000	18232	106287	88055
apache_clear_unsafe	SV	1.000	100000	0.9990	311	19244	18933
apache_md5_unsafe	MVAFR	0.016	1000	0.9990	15704	16138	434
apache_stringUtils_unsafe	MVAF	1.000	1000	0.9990	461	2732	2271
authMeReloaded_unsafe	MVAFR	0.025	10000	0.9990	197046	198718	1672

secret-dependent branches in order to obtain comparable results.

We also evaluated our approach on the unsafe program variants in order to explore how JIT can impact timing side channels already present at the source code level. In most cases, the secret-dependent branches of the safe variants can be matched to corresponding branchs in their unsafe variants. Hence, we evaluate the separability of the same ($\vec{I}_{b^+}$, $\vec{I}_{b^-}$) pairs across the safe and unsafe variants. This allows us to evaluate the appearance of the "same" timing side channel across both versions. Likewise, we used the same priming input across matching variants. We note that the public values chosen are not necessarily those which maximize the existing side channel in the unsafe variants. In fact, the ($\vec{I}_{b^+}$, $\vec{I}_{b^-}$) pairs do not even

Table 4.5: Experimental results for worst leakage for Blazer (top), Themis (middle) and DiffFuzz (bottom) unsafe variants.

Program variant	**Inst. pattern**	**Best leakage**					
		leakage	n	α	avg. time (ns)		
					I_{b+}	I_{b-}	diff
array_unsafe	SV	0.148	100000	0.5000	255	240	15
loopBranch_unsafe	SV	0.457	10000	0.9000	707	253	454
notaint_unsafe	SV	0.868	100000	0.9990	244693159	16331	244676828
sanity_unsafe	SV	0.111	10000	0.9990	5987	233	5754
straightline_unsafe	SV	0.148	100000	0.7000	201	199	2
unixlogin_unsafe	MVAF	0.160	10000	0.5000	213	223	10
modPow1_unsafe	MVAT	0.104	1000	0.5000	3250	3300	50
modPow2_unsafe	MVAT	0.126	100000	0.5000	20512	20595	83
pwdEqual_unsafe	MVAF	0.185	1000	0.5000	562	520	42
pwdEqual_unsafe	MVAT	0.189	100000	0.9000	419	421	2
pwdEqual_unsafe	SV	0.202	10000	0.9500	375	366	9
k96_unsafe	MVAT	0.176	10000	0.7000	11460	10886	574
gpt14_unsafe	MVAT	0.099	10000	0.7000	4233	4641	408
login_unsafe	SV	0.443	1000	0.5000	680	526	154
bootauth_unsafe	MVATR	0.022	1000	0.5000	14085	14169	84
bootauth_unsafe	MVAFR	0.044	1000	0.5000	12084	12282	198
jdk_unsafe	MVATR	0.006	1000	0.5000	3173	3963	790
jdk_unsafe	MVAFR	0.007	1000	0.5000	1019	1032	13
jetty_unsafe	SV	0.740	500000	0.9000	457	894	437
jetty_unsafe	MVAFR	0.033	1000	0.5000	765	771	6
jetty_unsafe	MVATR	0.002	10000	0.5000	306	322	16
orientdb_unsafe	MVATR	0.024	1000	0.5000	531	559	28
orientdb_unsafe	MVAFR	0.038	10000	0.5000	363	395	32
picketbox_unsafe	MVATR	0.013	1000	0.5000	448	487	39
picketbox_unsafe	MVAFR	0.063	1000	0.5000	616	641	25
spring_unsafe	SV	0.012	1000	0.5000	2479	2563	84
tomcat_unsafe	SVR	0.810	1000	0.5000	17496	208251	190755
pac4j_unsafe	SVR	0.498	10000	0.5000	20891	32312	11421
apache_clear_unsafe	SV	0.181	10000	0.5000	329	426	97
apache_md5_unsafe	MVAFR	0.009	1000	0.5000	14082	14197	115
apache_stringUtils_unsafe	MVAF	0.341	10000	0.5000	505	920	415
authMeReloaded_unsafe	MVAFR	0.003	1000	0.9000	203260	207381	4121

necessarily correspond to the timing channels originally present in the unsafe variant. The experiments on the unsafe versions are not aimed at finding the strongest side channels, but are shown as a comparison for the behavior found in the safe variants. The nosecret_safe, bootauth_safe, jdk_safe, orientdb_safe, picketbox_safe and authMeReloaded_safe variants have no secret-dependent branches and the apache_md5_safe and apache_salted_safe variants have no secret-dependent branches where both evaluations of the branch are possible. Therefore, we performed our instrumentation and fuzzing technique on their unsafe variants to generate ($\vec{I}_{b+}$, $\vec{I}_{b-}$) and I_π in order to evaluate those programs.

Table 4.6: Size of experimental subjects

(a) Dataset information for Blazer

Program variant	Program size (LOC)
array_safe	17
loopBranch_safe	27
nosecret_safe	5
sanity_safe	18
straightline_safe	13
unixlogin_safe	53
modPow1_safe	14
modPow2_safe	17
pwdEqual_safe	18
k96_safe	27
gpt14_safe	12
login_safe	? 21

(b) Dataset information for Themis

Program variant	Program size (LOC)
bootauth_safe	74
jdk_safe	36
jetty_safe	77
orientdb_safe	134
picketbox_safe	208
spring_safe	41
tomcat_safe	100
pac4j_safe	104

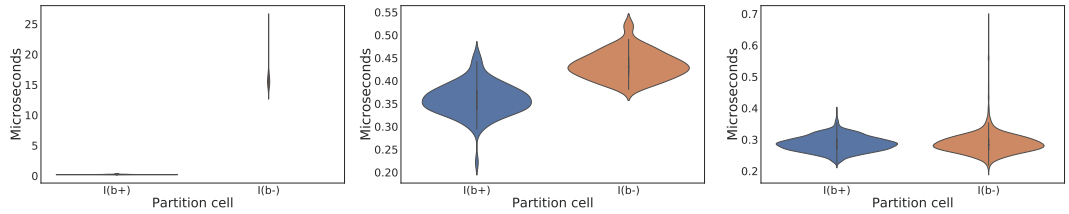
(c) Dataset information for DiffFuzz

Program variant	Program size (LOC)
apache_clear_safe	186
apache_md5_safe	159
apache_saltedPW_safe	191
apache_stringUtils_safe	29
authmreloaded_safe	40 ?

4.4.1 Experimental Results and Discussion

The results of our experiments on the "safe" Blazer, Themis and DiffFuzz variants are given in Tables 4.1, 4.2, 4.3. We report the name of the variant under test and the partition template matched. We report the highest leakage we observed, the parameters (priming number and distribution) leading to that leakage, and the average execution times of the program on input values from the sets I_{b^+} and I_{b^-} and the difference between these average execution times. We report the average execution times to give a sense of the observability of the resulting side channel. This is a consideration for an attacker with more limited timing capabilities and provides a more refined characterization of the side channel. Blazer, Themis, CoCo-Channel and DiffFuzz all report a static variant of the maximal cost difference across program paths; what we report is the runtime counterpart, which is a more realistic measure for observability.

We additionally report two other scenarios. First, we report the leakage and difference in execution time when JIT is disabled. These results provide a base line and are used to evaluate to what degree JIT optimization is responsible for any timing side channels as opposed to secret-dependent timing imbalance in the source code. Then, we report the maximum leakage and difference in execution time under what we call the "limited" priming scenario. In this scenario, the number of priming iterations is capped at 10,000 and the strongest distribution



(a) array_safe under Best Priming

(b) array_safe under Limited Priming

(c) array_safe with JIT disabled

Figure 4.4: Execution time distributions for the array_safe variant under the Best Priming, Limited Priming and without JIT.

considered is 0.9. This scenario gives us a sense of how well an attacker with limited capabilities could leverage JIT to induce side channels.

Our automated framework was able to induce large timing side channels in all "safe" program variants in the Blazer dataset with the exception of notaint_safe. These results contradict the results reported by four state-of-the-art analysis tools. We were able to induce large timing side channels in the Themis variants jetty_safe and spring_safe and the DiffFuzz variants apache_clear_safe and apache_stringUtils_safe. We also observed timing side channels in tomcat_safe and pac4j_safe, though we found these side channels to be present (though mitigated) even when JIT is disabled. The remaining safe Themis variants along with DiffFuzz authMeReloaded_safe and Blazer variant notaint_safe do not contain any secret dependent branches. The DiffFuzz variants apache_md5_safe and apache_saltedPW_safe contain a secret-dependent branch but only one evaluation of that branch is satisfiable.

Our results are validated at runtime, demonstrating the existence of the timing channels. In many cases, the conditional entropy reported is close to 1, meaning that an attacker would be able to deduce the membership of the secret input with almost certainty after one timing observation. Additionally, the difference in the average execution time of input from I_{b^+} and those from I_{b^-} can be on the order of tens of microseconds even for Blazer variants where the largest safe variant has only 20 basic blocks.

There are two scenarios in which no side channel was found in the safe variants: 1) when there is no secret dependent branch and 2) when any secret-dependent branch always evaluates to the same result. For password checking functions, safe variants were created by replacing secret-dependent branches with bit-wise manipulation code and, hence, these safe variants are resilient to JIT-induced side channels. Our observations point to potential mitigation techniques against JIT-induced side channels.

Our results for the case when JIT is disabled confirm that JIT compilation is responsible for the timing side channels. In almost all cases, disabling JIT eliminates the timing side channel and the information leakage. In loopBranch_safe, a small timing channel is reported. This is consistent with source-code-level analysis which reports a small imbalance between the relevant program paths. The small magnitude of this side channel implies that it is not likely to be observable in scenarios where our much larger JIT-induced timing channels would be. A timing side channel is also present in modPow2_safe. We believe this is because of slightly different multiplication done across the relevant paths which may vary in cost. Similar reasoning explains the timing side channel in apache_stringUtils_safe, where a variable cost instruction is likely the cause of the smaller side channel observed. We believe the side channel in pwdEqual_safe (MVAT) might be due to CPU level branch prediction. Finally, the small side channel reported in login_safe is likely due to the small number of samples taken (100) relative to the noise of the program.

As mentioned earlier, tomcat_safe and pac4j_safe both contain side channels even when JIT is disabled. In the case of tomcat_safe, this is congruous with the observation made by both the Themis and CoCo-Channel authors that the side channel is mitigated in the safe variant but not removed. Significant non-trivial computation is performed in this application and it is possible that the static models used in previous tools did not capture the program cost realistically. Likewise, tomcat_safe involves error handling code concerning interactions with an external database and it is likely that the static models for these computations were

approximate.

Our results in the Limited Priming Scenario are also informative. In all cases except apache_string_safe, the amount of leakage about the membership of the secret input is higher than the scenario when JIT is disabled. In some cases, such as jetty_safe, the resulting timing side channel is significantly smaller than the Best Priming Scenario and only tens of nanoseconds separate the timings of input from the partition cells. In other cases such as k96_safe, however, a timing side channel of tens of thousands of nanoseconds separates the timing distributions. Thus, even an attacker with more limited capabilities may be able to leverage JIT-induced side channels.

The results of our experiments on the unsafe Blazer, Themis and DiffFuzz variants are given in Table 4.4 and Table 4.5. We do not report results for the apache_saltedPW_unsafe variant of DiffFuzz since we do not find any input matching one of our partition patterns. Again, we do not aim to find the strongest side channel in the unsafe variant but to evaluate the impact of different priming strategies on the behavior of side channels. For example, our results on bootauth_unsafe show that a powerful side channel present in some JVM states all but disappears in others. In some cases, most notably modpow1_unsafe and modpow2_unsafe, no extremely strong side channel is detected. As mentioned previously, the public input values used in these experiments are not guaranteed to be values for which the intended side channel is strong in the unsafe variants. Additionally, the programs do differ from their safe variants so a priming strategy successful in inducing a timing side channel in the safe version might not be in the unsafe one. We believe this explains the lower leakage for some programs. Overall, our results on the unsafe variants further demonstrate the significant impact the runtime can have over timing side channels.

Coverage of Secret-Dependent Branches. In a few safe variants, additional branches were marked as secret dependent. In both loopBranch_safe and login_safe, an additional branch was marked but, as in the apache_md5_safe and apache_saltedPW_safe variants, the condition

associated with the branch could only ever be evaluated one way. Any path with the opposite evaluation is unrealizable. Therefore, there are no results reported for these cases. k96_safe, gpt14_safe, modpow1_safe modpow2_safe and apache_stringUtils_safe all have a branch condition nested within a loop but we only report results for either the MVAT or MVAF pattern. The other pattern is not applicable as the branch condition must evaluate to true (or false in the case of apache_stringUtils_safe) at least once. unixlogin_safe also marks a branch condition inside a loop as secret dependent, but only the MVAF pattern is evaluated. Neither the input generation responsible for (I_{b^+}, I_{b^-}) or AFL could find input satisfying the MVAF pattern. In this case, such an input exists but is very specific. Running AFL for longer or providing better seed input might enable it to find the input. Evaluation of apache_md5_unsafe and authMeReloaded_unsafe is limited to MVAFR for the same reason.

Importance of Priming Input. Table 4.1 reports the leakage values for the best priming input i_π per variant. A more nuanced look at the experiments is needed to understand the impact of the choice of priming value. In some cases, such as straightline_safe, all priming input values generated by KELINCI were effective at introducing a timing side channel. This simple program has only one secret-dependent branch, and the timing side channel is introduced regardless of which of the two program paths is favored. In other cases, such as pwdEqual_safe (MVAT), only a single priming input effectively introduced a timing side channel.

Relation of (I_{b^+}, I_{b^-}) to the secret input. How much danger an attacker being able to separate (I_{b^+}, I_{b^-}) poses can be answered in part by considering the size of the (I_{b^+}, I_{b^-}) relative to the entire input domain. In some programs, such as straightline_safe and array_safe, $I_{b^+} \cup I_{b^-}$ is the entire secret domain for the fixed public input. In others, such as the MVAF pattern of jetty_safe, (I_{b^+}, I_{b^-}) is the entire domain of possible secret input of a fixed length (we chose length 4). This means that if an attacker is aware that the secret input is of length 4, she will be able to determine the membership of that input. Another part of the answer depends on the relative sizes of (I_{b^+}, I_{b^-}) to each other. In sanity_safe, for instance, the set of

possible secret inputs is evenly divided between ($I_{b^+}$and I_{b^-}). However, in the MVAT pattern of jetty_safe, the only string of length 4 always matching a public string of length 4 is that public string itself. Therefore, answering the membership query is equivalent to determining whether or not the secret is a certain exact string. There are certainly applications where even this is dangerous (suppose a branch condition allows you to learn if an anonymous user is in fact a particular celebrity); however, less information about the secret itself can be learned in this case. Ultimately, we demonstrate the *existence* of *a* side channel. This is the same question that Blazer, Themis, CoCo-Channel and DiffFuzz concern themselves with. The follow up question of how much information the side channel leaks about the secret itself is left to other analyses.

Potential Patches. Our results indicate that any JIT-enabled system is vulnerable to a timing side channel arising from biased input distributions. One potential patch is to disable JIT. However, doing so critically impacts the performance of the JVM. For example, the execution time of one call to login_safe is under 500 nanoseconds with JIT optimization and *over 3 seconds* when JIT is disabled. Disabling JIT slows this program by a factor of more than 7000000 times. Such performance loss would render Java programs unusable for many applications. Selectively disabling JIT only for highly sensitive, security-critical code would mitigate this performance loss but ultimately results in a trade-of between performance and security that needs to be reconciled. Another potential option would be replace just-in-time compilation with ahead-of-time compilation (AoT) [92], where all optimization and code generation is done *before* running the program and runtime statistics are not used in optimization decisions. It is thus likely to be more robust against side channels introduced at runtime and this additional security could be another reason to adopt it for certain security-critical methods. Finally, our results on variants such as notaint_safe motivates replacing secret dependent branches with bit-wise manipulation code as a mitigation.

4.5 Conclusions

We demonstrated that timing side channels in software are not a static phenomenon but can be introduced dynamically into programs through biased input distributions. In particular, the just-in-time (JIT) compilation mechanism, which is critical for the performance of Java programs, can be leveraged to introduce timing channels into programs that do not contain side-channel vulnerabilities when JIT is disabled. We have developed and evaluated an automatic technique to fuzz the JVM in order to detect JIT-induced side channels. We use this technique to show that the previously safe labeled variants of the well-known and studied benchmarks are vulnerable to JIT-induced timing side channels, contradicting the results of four state-of-the-art analysis tools. We conclude that JIT-induced side channels are prevalent and the side channel vulnerability detection techniques have to take into account the impact of a program's runtime during side channels analysis.

Chapter 5

Mitigation of JIT-Induced Side Channels

5.1 Introduction

In the last chapter, I've shown that JIT-induced side channels can lead to information leaks in software. In particular, JIT compilation can introduce side channels even into source code that has been carefully balanced by a developer. A concern therefore arises about how these side channels might be effectively mitigated. In this chapter, I develop and evaluate mitigation strategies for JIT-based side channels.

My mitigation techniques target the runtime. As discussed, the runtime can impact the execution time of a program in nuanced and complicated ways, potentially giving rise to timing side channels. Therefore, mitigation at the runtime level promises a higher level of robustness than mitigation at the source code level. In fact, common source-code level mitigation strategies, such as branch balancing, have already been shown as ineffective at preventing side channels from arising due to runtime behavior.

I explore multiple runtime-level strategies for JIT-induced side channel mitigation, focusing

on the widely used Java runtime, OpenJDK. First, I define three different compilation strategies and empirically evaluate their effectiveness at mitigating JIT-based side channels while retaining high performance. Then I introduce more nuanced algorithms for JIT-induced side channel mitigation and present simulation results arguing for their effectiveness at both preventing leakage as well as achieving high performance.

The rest of this chapter is organized as follows: In Section 5.2, I present different compilation strategies aimed at mitigating JIT-based side channels. Then I experimentally evaluate the effectiveness of these strategies on benchmarks from the Java standard library in Sections 5.3 and 5.4. In Section 5.5, I present additional, more refined mitigation strategies and argue for their effectiveness. Finally, I conclude in Section 5.6.

5.2 Command-Line Compilation Strategies

In this section, I present three different compilation strategies to mitigate JIT-based side channels. Each of these strategies can be implemented using command line options in OpenJDK.

The NOJIT Strategy

The most straightforward technique for mitigating JIT-induced side channels is simply disabling all just-in-time compilation. With JIT disabled, there is no optimization based on dynamic profiling. Therefore a bias in the input distribution of the program cannot lead to the runtime environment favoring certain program behavior.

Leakage Analysis Under NOJIT no JIT-based side channels arise. Timing side channels more generally may still be present, whether through imbalances at the source code level or through CPU or hardware mechanisms such as caching. We use NOJIT as a baseline to determine the extent that a side channel arises due to JIT compilation.

Performance Analysis Disabling JIT could lead to significant performance loss. Many modern programming languages such as Java and Javascript rely on just-in-time compilation for performance [93]. The degree to which performance is impacted will depend on the target application but in some cases, disabling JIT entirely could leave systems unusable.

The DisableC2 Strategy

Different levels of optimization give rise to timing side channels of differing strength and observability. In OpenJDK, there are basically two types of compilers. The client compiler, or C1 compiler, is a fast, lightly optimizing bytecode compiler while the C2 compiler, is a highly optimizing bytecode compiler that fully leverages profiling information. Some C2-level optimizations include conditional constant type propagation, constant folding, algebraic identities, method inlining (aggressive, optimistic, and/or multi-morphic), intrinsic replacement, loop transformations (unswitching, unrolling), array range check elimination. [94]. Optimistic compilation is a C2-level optimization meaning that for Topti to apply, the function under test must be compiled to the C2-level. Therefore, we propose and evaluate the DisableC2 strategy, where we restrict the runtime to only C1-level optimizations.

Leakage Analysis Under DisableC2 no C2-level optimizations occur. This means that no timing side channels will arise under Topti. Additionally, timing side channels arising due to the difference in performance of a function compiled under C2 compilation versus under C1 compilation (which can occur under Tmeth, Tmeth-ii or Tmeth-be) will not occur. However, there may still be timing side channels arising due to the difference in performance of a function compiled under C1 versus interpreted. DisableC2 does not protect against timing side channels arising under Tbran.

Performance Analysis Disabling C2 compilation will effect different programs and workloads differently. In general, programs expected to run for long periods of time will benefit from

more aggressive C2 optimizations and will experience negative impacts to performance under DisableC2. On the other hand, programs which benefit more from a quicker start-up than aggressive compilation will not experience slowdown under DisableC2.

The MExclude Strategy

Not all components of a program or application necessarily handle sensitive input. Therefore one mitigation strategy is to disable JIT compilation, not for an entire program, but for the method m specified as vulnerable by the vulnerability template. Inspired by this, we propose and evaluate the MExclude strategy.

Leakage Analysis The MExclude strategy protects against any side channel caused by the compilation of m. There are side channels of all templates (excepting Tmeth and Tmeth-ii) that fall into this character. However, not all JIT-based side channels are captured by this categorization. For example, a timing side channel can arise under Tmeth due to the compilation of a callee method, not the compilation of m itself.

Performance Analysis Performance under MExclude varies based on characteristics of m and the workload. When calls to m constitute a large portion of the workload or when m is an expensive function, performance is more negatively impacted.

5.3 Experimental Setup

In this section, we evaluate four different compilation strategies: a baseline strategy which is the default compilation configuration of OpenJDK 14 and the three mitigation strategies outlined in Section 5.2. We determine how successful each strategy is in mitigating JIT-induced side channels and how much of a performance loss each incurs.

Hardware Setup The library experiments were run on Ubuntu 16.04 machine with Intel i5 3.5GHz X4 processors and 128GB of memory, with a Linux 4.4.0-81 64-bit kernel, and Java 14 Platform Standard Edition version 1.14.0 (14-internal) from OpenJDK.

Implementation We used command line arguments to implement each strategy. We disable JIT compilation as per the NOJIT strategy using the option "-Xint". We disable C2-level compilation as per the DisableC2 strategy using the option "-XX:TieredStopAtLevel=3". We disable compilation for a particular method as per the MExclude strategy using the option "-XX:CompileCommand=exclude,the/package/and/Class,methodName". The methods disabled for each benchmark are given in the "Method name" column of the pertinent results table.

Benchmarks We evaluate each compilation strategy on the set of built-in Java functions from Chapter 3 under the IPM and NPM-LAB attack models. The built in Java functions are drawn from the **java.math.BigInteger**, **java.lang.Math** and **java.lang.String** classes. While we used classes from OpenJDK 8 in Chapter 3, we now use the OpenJDK 14 versions [95] of these functions.

We use every IPM and NPM-LAB benchmark from Chapter 3. We also augment our evaluation with new experiments. Each method marked as vulnerable under the Tbran and Tmeth templates is also vulnerable under the Topti template (for the same branch instruction). Therefore we extend the benchmark by additionally evaluating those methods under Topti. We generate priming amounts and ratios for these benchmarks as outlined in Chapter 3. As before, ϕ is the predicate over input susceptible to being leaked according to the vulnerability template.

Leakage Evaluation We evaluate leakage under each compilation strategy as follows. First, we follow the same experimental procedure as in Chapter 3 for the baseline compilation strategy. We use the same priming values, one satisfying the predicate ϕ and one satisfying $\neg\phi$ as well as the same two sets of possible secret inputs. As there have been changes from OpenJDK 8

to OpenJDK 14, we repeat our procedure to find the appropriate priming amount and ratio for each benchmark. Then we perform the appropriate IPM or NPM-LAB experiment, each for 1,000 iterations.

We evaluate the other compilation strategies using these priming parameters, priming inputs and sets of secret input. To evaluate the DisableC2 strategy, we replicate the baseline experiment but include the "-XX:TieredStopAtLevel=3" command line option. For evaluation of the MExclude strategy, we include the "-XX:CompileCommand=exclude,the/package/and/Class,methodName" command line option. Finally, we evaluate NOJIT as in the previous chapter using a small, fixed, balanced amount of priming (50 calls on both sides) to avoid artificial noise from initial class/method loading delays.

We again use conditional entropy to quantify the information leaked. A value of 0.0 means no leakage, while 1.0 means full leakage of ϕ's value from one timing observation.

Performance Evaluation Ideally, we want to mitigate JIT-induced side channels while still preserving performance. This leads us to additionally evaluate each mitigation strategy in terms of the performance loss it incurs. Since this may depend on how the benchmark in question is used, we evaluate the performance of each mitigation strategy under three different workloads: for 1,000, 10,000 and 100,000 function calls. In these experiments, the program input is divided evenly between input that satisfies the predicate and those that do not. While we do not exhaustively evaluate possible workloads, these three provide us with a sense of how the performance changes as the benchmark is used more.

5.4 Experimental Results

Leakage results for the IPM and NPM-LAB benchmarks are given in Table 5.1 and Table 5.2 respectively and performance results for each benchmark are given in Table 5.3. In this section,

we analyze the experimental data, present out results and discuss interesting phenomena.

The NOJIT Strategy With JIT compilation disabled, there is no leakage arising due to dynamic compilation. Nevertheless some functions (such as BigInteger.shiftLeft still contain timing side channels introduced at the source code level. None of our compilation strategies are aimed at eliminating these kinds of side channels vulnerabilities so, while we note their appearance, we do not consider it a failure when our mitigation strategies do not remove them.

As expected, disabling JIT compilation can greatly impact performance. For very light workloads (1,000 function calls), the impact of disabling JIT is negligible. In fact, the start up cost of just-in-time compilation can even outweigh the benefits of compilation. As the size of workloads increase, the impact of disabling JIT is more greatly felt. Even for these fairly small built-in Java functions, disabling JIT can slowdown the runtime of workloads by over 13x. For more complex and expensive programs, the impact on performance would likely be even larger.

The DisableC2 Strategy Disabling only C2-level compilation results in partial mitigation of JIT-induced side channels. Because optimistic compilation is a C2-level optimization, timing side channels arising due to Topti are mitigated. The low leakage results for the Topti benchmarks BigInteger.min, BigInteger.valueOf, Math.max, Math.nextAfter can be ascribed to this. Nevertheless, there is still significant leakage in the Topti benchmarks BigInteger.shiftLeft and Math.ulp. Manual inspection shows that these leakages are not due to optimistic compilation. Instead both benchmarks contain source code level side channels. These timing side channels actually become more reliable as the function is optimized to the C1-level as there is less noise.

The DisableC2 strategy is less effective at mitigating timing side channels arising under Tmeth and Tbran. Side channels due to Tmeth do not appear to have been meaningfully mitigated at all. This is because the difference between performance of a C1-compiled function

and the interpreted version of the same function is observable. Some timing channels arising due to TBRAN (BigInteger.valueOf) appear to be partially mitigated while others (Math.max) do not. This is likely due to whether the option for higher compilation level results in a more steady-state execution time behavior with less noise obscuring the side channel.

The DISABLEC2 strategy introduces significantly less performance loss than the NOJIT strategy. Nevertheless as the size of the workload increases, the benefits of tiered compilation becomes more apparent. In general; for larger applications, there can be significant benefit to server-level compilation [96, 97]. It is likely that the fairly small and non-complex Java benchmarks we experiment do not benefit significantly from aggressive, server-level compilation, yielding a smaller loss of performance under the DISABLEC2 strategy than we can expect for larger applications.

The MEXLUDE Strategy Mitigation results for the MEXLUDE strategy are also mixed, but overall the strategy appears to be less successful than the DISABLEC2 strategy. In some cases, such as TOPTI benchmark BigInteger.min leakage is actually increased. In this particular case, the reason for the increased leakages lies in an uncommon trap in a callee function of BigInteger.min. Even though BigInteger.min isn't compiled, the callee function is, causing the strong side channel reported. This emphasizes the general problem with the MEXLUDE strategy. It is hard to guarantee without detailed inspection if disabling compilation of the method m is enough to eliminate the side channel.

The MEXLUDE strategy also negatively impacts performance. Our results in Table X show that, for longer workloads, the runtime performance under the MEXLUDE strategy can be degraded by over 4x. Combined with its mixed results for mitigating leakage, this performance argues against the MEXLUDE strategy as a general solution for mitigating JIT-induced side channels. It is important to note; however, that in our experiments the method for which compilation is disabled constitutes a large portion of the function calls in the workload. This

may not always be the case, meaning that in other scenarios this strategy could achieve better performance. In general though, without a nuanced understanding of the program or application under test, this strategy does not seem well suited to mitigation.

Nevertheless, the intent behind the MExclude strategy does show promise. If we had a more systematic and accurate way of deciding which methods or code segments should be excluded from compilation due to privacy concerns, this strategy might more successfully mitigate leakage with an acceptable performance loss for some workloads.

Changes versus OpenJDK8 The priming amount and ratios for some benchmarks differ from those in Chapter 3. This is due to the change in the runtime under test – we are now using the more recent OpenJDK 14 instead of OpenJDK 8. Therefore, the priming parameters vary slightly. Even so, the templates previously introduced allowed us to find new priming values efficiently.

String Functions We did not find much success in inducing JIT-based side channels in the built-in **java.lang.String** functions, even under the Baseline strategy. This stands in contrast to the success we achieved in Chapter 3. We looked into this curious phenomena and found that in OpenJDK 14, many string functions are marked as hotspot intrinsic candidates [95]. An intrinsic is a function whose implementation is handled specially by a compiler and may have one or more optimized implementations on some platforms. The implementations of intrinsic methods tend to be machine-dependent, well-optimized instructions. Because the runtime is using this intrinsic function implementation, these string functions are not being handled by the optimizing compiler in the same way as before. We believe this accounts for the marked change in side-channel behavior for these functions.

IPM versus NPM-LAB Results As before, there is less leakage overall under the NPM-LAB model, where the observer is more restrained about what observations they may make. Moreover,

Table 5.1: Leakage under IPM

Method name	Template applied	Priming amount	Priming ratio	Leakage BASELINE	Leakage NOJIT	Leakage DISABLEC2	Leakage MEXLUDE
BigInteger.min	TOPTI	100,000	0.998	0.58	0.03	0.02	1.00
BigInteger.valueOf	TBRAN	10,000	0.900	0.86	0.04	0.20	0.32
BigInteger.valueOf	TOPTI	100,000	0.998	0.83	0.05	0.01	0.19
BigInteger.shiftLeft	TMETH	5,000	0.950	0.92	0.73	0.95	0.62
BigInteger.shiftLeft	TOPTI	100,000	0.998	0.78	0.64	0.90	0.95
Math.max	TBRAN	1,000	0.900	0.34	0.05	0.31	0.38
Math.max	TOPTI	100,000	0.998	0.81	0.06	0.02	0.03
Math.ulp	TMETH	10,000	0.950	1.00	0.22	0.91	0.53
Math.ulp	TOPTI	100,000	0.998	0.99	0.21	1.00	0.26
Math.nextAfter	TOPTI	100,000	0.998	0.86	0.81	0.02	0.60
Math.min	TOPTI	100,000	0.998	0.03	0.05	0.02	0.02
String.equals	TOPTI	100,000	0.998	0.03	0.08	0.02	0.03
String.compareTo	TOPTI	100,000	0.998	0.08	0.34	0.02	0.80
String.startsWith	TBRAN	1,000	0.900	0.32	0.12	0.19	0.32
String.startsWith	TOPTI	100,000	0.998	0.03	0.23	0.02	0.04

because all leakage under NPM-LAB is due to optimistic compilation, the DISABLEC2 strategy is successful in mitigating these side channels. This success shows that the challenge of mitigation may be made simpler in cases where only certain attack models are concerns.

5.5 Refined Mitigation Strategies

In this section, we present additional mitigation strategies inspired by our results for the command-line compilation strategies. Empirically evaluating these more nuanced mitigation strategies would require modifying OpenJDK itself, a challenging implementation task. Therefore, we argue for their effectiveness by simulation, mathematically evaluating their impact on both leakage and performance.

A key observation from the last chapter is that optimistic compilation is responsible for timing side channels are the largest magnitude and also for all the side channels we observed arising under NPM-LAB. Therefore, we propose two new mitigation strategies aimed at elimi-

Table 5.2: Leakage under NPM-LAB

Method name	Template applied	Priming amount	Priming ratio	Leakage BASELINE	Leakage NOJIT	Leakage DISABLEC2	Leakage MEXLUDE
BigInteger.min	TOPTI	100,000	0.998	0.52	0.03	0.02	0.62
BigInteger.valueOf	TBRAN	10,000	0.90	0.07	0.31	0.06	0.15
BigInteger.valueOf	TOPTI	100,000	0.998	0.68	0.36	0.01	0.04
Math.max	TBRAN	10,000	0.90	0.03	0.03	0.04	0.03
Math.max	TOPTI	100,000	0.998	0.02	0.03	0.02	0.02
Math.nextAfter	TOPTI	100,000	0.998	0.80	0.06	0.01	0.04
Math.min	TOPTI	100,000	0.998	0.03	0.04	0.03	0.02
String.equals	TBRAN	10,000	0.90	0.03	0.04	0.04	0.03
String.equals	TOPTI	100,000	0.998	0.03	0.04	0.02	0.03
String.compareTo	TOPTI	100,000	0.998	0.08	0.04	0.01	0.22
String.startsWith	TBRAN	1,000	0.90	0.04	0.04	0.03	0.04
String.startsWith	TOPTI	100,000	0.998	0.03	0.04	0.03	0.03

Table 5.3: Performance Results for all command-line strategies

Method name	Priming amount	AVG Time (ms) BASELINE	AVG Time (ms) NOJIT	Speedup vs BASELINE	AVG Time (ms) DISABLEC2	Speedup vs BASELINE	AVG Time (ms) MEXLUDE	Speedup vs BASELINE
BigInteger.min	1000	0.28	0.23	0.85x	0.28	1.03x	0.28	1.02x
BigInteger.min	10000	0.92	2.29	2.50x	0.83	0.91x	1.16	1.27x
BigInteger.min	100000	4.11	22.93	5.58x	4.61	1.12x	8.23	2.00x
BigInteger.valueOf	1000	0.81	0.80	0.99x	0.81	1.00x	0.81	1.00x
BigInteger.valueOf	10000	1.24	1.39	1.12x	1.22	0.98x	1.48	1.19x
BigInteger.valueOf	100000	3.77	7.29	1.93x	4.02	1.07x	8.23	2.19x
BigInteger.shiftLeft	1000	1.05	1.09	1.04x	1.09	1.04x	1.08	1.03x
BigInteger.shiftLeft	10000	8.38	10.48	1.25x	8.23	0.98x	9.68	1.16x
BigInteger.shiftLeft	100000	55.99	82.84	1.48x	59.32	1.06x	72.18	1.29x
Math.max	1000	0.09	0.08	0.88x	0.10	1.06x	0.09	0.97x
Math.max	10000	0.51	0.67	1.33x	0.51	1.00x	0.75	1.48x
Math.max	100000	3.01	6.56	2.18x	3.41	1.13x	7.44	2.47x
Math.ulp	1000	0.30	0.27	0.92x	0.30	1.00x	0.28	0.94x
Math.ulp	10000	0.77	2.60	3.37x	0.71	0.92x	1.38	1.79x
Math.ulp	100000	3.36	25.54	7.61x	3.75	1.12x	10.60	3.16x
Math.nextAfter	1000	0.13	0.12	0.90x	0.14	1.05x	0.13	0.99x
Math.nextAfter	10000	0.57	1.02	1.79x	0.58	1.01x	1.14	2.00x
Math.nextAfter	100000	3.09	9.94	3.22x	3.45	1.12x	11.09	3.59x
Math.min	1000	0.05	0.08	1.68x	0.04	0.84x	0.08	1.63x
Math.min	10000	0.35	0.70	1.99x	0.34	0.97x	0.72	2.05x
Math.min	100000	2.72	6.81	2.50x	2.85	1.05x	7.15	2.63x
String.equals	1000	0.03	0.14	3.97x	0.05	1.31x	0.09	2.72x
String.equals	10000	0.34	1.36	3.99x	0.44	1.28x	0.91	2.67x
String.equals	100000	2.92	13.62	4.67x	3.91	1.34x	8.79	3.01x
String.compareTo	1000	0.39	0.56	1.46x	0.37	0.95x	0.38	0.98x
String.compareTo	10000	1.14	5.56	4.89x	1.08	0.95x	1.56	1.37x
String.compareTo	100000	4.11	55.62	13.52x	5.04	1.22x	11.82	2.87x
String.startsWith	1000	0.07	0.52	7.86x	0.07	1.00x	0.23	3.47x
String.startsWith	10000	0.64	5.20	8.15x	0.63	0.99x	2.22	3.48x
String.startsWith	100000	4.57	51.86	11.35x	5.70	1.25x	21.64	4.74x

nating side channels arising under TOPTI. Since the DISABLEC2 strategy achieved good success at eliminating side channels arising under TOPTI, we use it as a starting off point for our refined strategies.

The NOOPTI Strategy

Side channels arising under TOPTI are caused by optimistic compilation. Therefore instead of disabling C2-level compilation entirely, we only disable optimistic compilation.

Leakage Analysis Optimistic compilation is disabled, therefore no side channels under TOPTI will arise.

Performance Analysis This strategy allows all C2-level optimizations except optimistic compilation. Therefore, it will not experience the same degree of performance loss as the DISABLEC2 strategy when confronted by larger workloads.

We first discuss the performance of the NOOPTI strategy under workloads where no optimistic assumption is ever broken. These are the workloads where the NOOPTI strategy has the worst performance when compared to the BASELINE strategy.

Performance Analysis: Compilation Time One benefit of optimistic compilation is to save time during compilation. Aggressive compilation can be expensive. So, if there is some program behavior that never or rarely occurs, the runtime may save on compile time by simply not opting to compile the code for that behavior. For example, if one side of a conditional is very rare, the optimizing compiler can reduce compilation time by not compiling that branch.

For any given method m and conditional b in method m, let $\text{CompTime}(m_{b_0})$ be the time taken to compile the IF branch of the conditional and $\text{CompTime}(m_{b_1})$ be the time taken to compile the ELSE branch of the conditional. Without optimistic compilation, both branches are compiled and $\text{CompTime}(m_{b_0}) + \text{CompTime}(m_{b_1})$ contributes to the total compilation

time. If optimistic compilation is enabled and the optimistic assumption made that one branch is never taken, then only either $\textsc{CompTime}(m_{b_0})$ *or* $\textsc{CompTime}(m_{b_1})$ is added to the total compilation time. Without loss of generality, we negate conditionals and re-order branches, so that the IF branch is always the common behavior and only $\textsc{CompTime}(m_{b_0})$ is added to the compilation time. $\textsc{CompTime}(m_{b_1})$ is therefore saved by enabling optimistic compilation. If the optimistic assumption is never broken and the function never recompiled, the end result is effective and less expensive compilation.

For a given method m and workload w, we use the notation $\Delta(\textsc{CompTime}(m, w))$ to be the difference in compilation time of m under workload w when optimistic compilation is disabled versus when optimistic compilation is enabled. As we consider only workloads where no optimistic assumption is ever broken, this number is never negative. $\Delta(\textsc{CompTime}(m, w))$ is proportional to the sum $\sum_{b \in B_m(w)} \textsc{CompTime}(m_{b_1})$ where $B_m(w)$ is the set of conditionals in m for which the optimizing compiler makes optimistic assumptions given the workload w. For program p, the compilation time saved by enabling optimistic compilation for a given workload w is $\sum_{m \in p} \Delta(\textsc{CompTime}(m, w))$.

Performance Analysis: Execution Time There can be additional benefits to optimistic compilation. Sometimes, such as in the case of explicit null pointer checks, the optimizing compiler not only removes the extremely rare conditional but also the conditional check itself [98, 99]. This can remove jumps in the assembly code. The end result is aggressively optimized code that is not only less expensive to generate but also more efficient when executing the common program behavior.

For any given method m and conditional b in method m, let $\Delta(\textsc{ExecTime}(m_b))$ be the difference in time taken to execute the common branch of conditional b when method m is compiled without and with optimistic compilation. $\Delta(\textsc{ExecTime}(m_b))$ ranges between 0 and the sum of a jump instruction and the time taken to evaluate the conditional b.

For a given method m and workload w, we use the notation $\Delta(\textsc{ExecTime}(m, w))$ to be the difference in execution time of m under workload w when optimistic compilation is disabled versus when optimistic compilation is enabled. As we consider only workloads where no optimistic assumption is ever broken, this number is never negative. $\Delta(\textsc{ExecTime}(m, w))$ is proportional to the sum $\sum_{b \in B_m(w)} \Delta(\textsc{ExecTime}(m_b))$ where $B_m(w)$ is the set of conditionals in m for which the optimizing compiler makes optimistic assumptions given the workload w. For program p, the execution time saved by enabling optimistic compilation for a given workload w is $\sum_{m \in p}$ $\Delta(\textsc{ExecTime}(m, w))$.

For any workload w where an optimistic assumption is never broken, the total performance loss under the NoOpti strategy is $\sum_{m \in p} \Delta(\textsc{CompTime}(m, w)) + \Delta(\textsc{ExecTime}(m, w))$.

Performance Analysis: De-Optimization Time On the other hand, in the event of workloads where an optimistic assumption is proven incorrect, the NoOpti strategy will not incur the same performance loss as the Baseline strategy. Under the Baseline strategy, when an uncommon trap is triggered, the runtime must deoptimize the culprit method and replace it with a slower, more conservative version. This less optimized version of m is used until m is again marked for recompilation and re-compiled.

For a method m and conditional b, let $\textsc{DeoptTime}(m_b)$ be the cost of the uncommon trap handling. For a given workload w, let $\Delta(\textsc{DeoptExecTime}(m, w))$ be the difference in execution time between the highly optimized version of m and the less optimized version of m the runtime must revert to after triggering an uncommon trap. Let $\textsc{ReoptTime}(m, w)$ be the time taken to recompile the method m under workload w. If m is never recompiled for workload w, $\textsc{ReoptTime}(m, w)$ is 0. For a program p, the total cost caused by the triggering of uncommon traps for a workload w is $\sum_{m \in p} \sum_{b \in B*_{m_w}} \textsc{DeoptTime}(m_b) + \Delta(\textsc{DeoptExecTime}(m, w)) +$ $\textsc{ReoptTime}(m, w)$, where $B*_{m_w}$ is the set of conditionals m for which the optimizing compiler makes optimistic assumptions later proven incorrect given the workload w. We refer to this

term as DeoptCost(m, w). Note that $B*_{m_w}$ is a subset of $B_m(w)$.

For any workload w where at least one optimistic assumption is broken, the total performance loss under the NoOpti strategy is $\sum_{m \in p} \Delta(\text{CompTime}(m, w)) + \Delta(\text{ExecTime}(m, w))$ - DeoptCost(m, w). Depending on whether the cost associated with triggering uncommon traps is greater than the compilation and execution time saved by use of the optimistic compilation mechanism, the NoOpti strategy might actually perform better than the Baseline strategy on workloads where an uncommon trap is triggered.

The ModOpti Strategy

Not all instances of optimistic compilation necessarily introduce timing side channels. In the ModOpti strategy, we augment our mitigation technique with taint analysis. Instead of disabling optimistic compilation all together, we only disable optimistic compilation for branches of conditionals marked as dependent on secret input.

Leakage Analysis Assuming our taint analysis is sound, then then no timing side channel under Topti will arise.

Performance Analysis The ModOpti strategy improves the performance of the NoOpti strategy by continuing to allow optimistic compilation on all predicates not dependent on secret input. The runtime can therefore save additional compilation time.

Performance Analysis: Compilation Time For a given method m, workload w and secret input s, $\Delta(\text{CompTime}(m, w, s))$ is proportional to the sum $\sum_{b \in B^s_m(w)} \text{CompTime}(m_{b_1})$ where $B^s_m(w)$ is the set of *secret-dependent* conditionals in m for which the optimizing compiler makes optimistic assumptions given the workload w. For program p, the compilation time saved by enabling full optimistic compilation versus the ModOpti strategy for a given workload w is $\sum_{m \in p} \Delta(\text{CompTime}(m, w, s))$.

Performance Analysis: Execution Time For a given method m, workload w and secret input s, $\Delta(\textsc{ExecTime}(m, w, s))$ is proportional to the sum $\sum_{b \in B^s_m(w)} \Delta(\textsc{ExecTime}(m_b))$ where $B^s_m(w)$ is the set of *secret-dependent* conditionals in m for which the optimizing compiler makes optimistic assumptions given the workload w. For program p, the execution time saved by enabling optimistic compilation versus the ModOpti strategy for a given workload w is $\sum_{m \in p}$ $\Delta(\textsc{ExecTime}(m, w, s))$.

For any workload w where an optimistic assumption is never broken, the total performance loss under the ModOpti strategy is $\sum_{m \in p} \Delta(\textsc{CompTime}(m, w, s)) + \Delta(\textsc{ExecTime}(m, w, s))$.

Performance Analysis: De-Optimization Time For a program p, the total cost caused by the triggering of uncommon traps for *secret-dependent* conditionals for a workload w is $\sum_{m \in p} \sum_{b \in B*^s_{m_w}}$ $\textsc{DeoptTime}(m_b) + \Delta(\textsc{DeoptExecTime}(m, w)) + \textsc{ReoptTime}(m, w)$, where $B^s*_{m_w}$ is the set of *secret-dependent* conditionals m for which the optimizing compiler makes optimistic assumptions later proven incorrect given the workload w. We refer to this term as $\textsc{DeoptCost}(m, w, s)$. Note that $B^s*_{m_w}$ is a subset of $B^s_m(w)$.

For any workload w, the total performance loss under the ModOpti strategy is $\sum_{m \in p}$ $\Delta(\textsc{CompTime}(m, w, s)) + \Delta(\textsc{ExecTime}(m, w, s))$ - $\textsc{DeoptCost}(m, w, s)$.

We believe that these two mitigation strategies will achieve high performance while eliminating side channel leakage arising under Topti.

5.6 Conclusions

In this chapter, I have introduced and evaluated a set of compilation strategies for their success at mitigating JIT-induced side channels while retaining high performance. I first demonstrated how command-line options could be used to define different Java compilation techniques. I evaluated these compilation techniques on a set of benchmark programs finding mixed success

for both their ability to mitigate and achieve high performance. Then I introduced two new mitigation strategies requiring modifications of the Java runtime. I analyzed these strategies for their ability to mitigate side channels arising under Topti without compromising performance and argued for their success.

Ultimately, the challenge of ensuring that the runtime does not introduce timing side channels is a difficult one. Concepts such as differential privacy and its extension to runtime systems could provide a means of specifying undesirable or privacy-violating runtime behavior. Not all JIT-induced side channels violate this extension of differential privacy (for example those arising under Tbran do not), so satisfying it does not guarantee the absence of JIT-induced side channels. However it could show the absence of a category of JIT-induced side channels deemed to be particularly dangerous or privacy-concerning.

Chapter 6

Quantitative Leakage Analysis with Constraint Normalization and Parameterized Caching

6.1 Introduction

In the previous chapters, we focused on determining *if* a predicate on secret input is leaked through a side channel vulnerability. How much information about secret input can be revealed by that predicate is still an open concern. Quantitative program analyses provide a means for understanding the capacity of side channels to leak sensitive information. Quantitative information flow uses information-theoretic measures to quantify how much information about a program's secret input can be expected to be learned from a side-channel observable [100, 101]. This information can be critical in determining the severity of a side-channel vulnerability.

The developments in the area of Satisfiability-Modulo-Theories (SMT) [102, 103] and the implementation of powerful SMT solvers [104–106] have been the key technological developments that enabled the rise of effective symbolic program analysis and testing techniques in the

last decade [107–110]. However, performing symbolic analysis via satisfiability checking is not sufficient for quantitative program analysis, which is an important problem that arises in many contexts such as probabilistic analysis [111–113], reliability analysis [114] and, as mentioned, quantitative information flow [101, 115–123]. The enabling technology for quantitative program analysis is model-counting constraint solvers. A model-counting constraint solver returns the number of solutions for a given constraint within a given bound [124–126].

Since constraint solving and model counting are heavily used in program analysis, improving performance of these tasks is of critical importance. In this chapter, I present a new approach for constraint normalization and constraint caching with the goal of improving the performance of quantitative program analyses.

The key step in constraint caching is normalization of constraints, i.e., reducing constraints to a normal form, where if two constraints are equivalent (w.r.t. satisfiability or model counting), then they are reduced to the same normal form. Using the normal form of a constraint as a key, we can recover results of previous satisfiability or model-counting queries without recomputing them.

Earlier techniques for constraint caching [127–129] 1) focus only on numeric constraints and do not handle string constraints, 2) use normalization techniques that preserve the exact solution set of a constraint, which reduces cache hits for model-counting queries, and 3) always produce cache misses for model-counting queries if a different bound is used, even if the queried constraint remains the same. In this chapter, I extend those earlier results in multiple directions:

- I present constraint normalization techniques for model counting where the normalized form of a constraint may not preserve the solution set for the constraint but preserves the cardinality of the solution set.
- I extend constraint caching to string constraints which is crucial for analyzing string manipulating code which may be vulnerable to side channels.

- I present a parameterized caching approach where, in addition to the result of a model-counting query, we also cache a counting object in the constraint cache that allows us to efficiently recount the models for different bounds.

- I present an extensible group-theoretic framework for constraint normalization that generalizes earlier results on constraint normalization for caching.

I implemented these techniques in the tool Cashew, which we integrated with Symbolic PathFinder (SPF) [130] and a model-counting constraint solver ABC [126]. Our experiments demonstrate that constraint caching can improve the performance of quantitative program analysis significantly.

The rest of the chapter is organized as follows: In Section 2 I provide some motivating examples for constraint caching. In Section 3 I give an overview of our constraint caching framework. In Section 4 I discuss our group-theoretic normalization scheme. In Section 5 I describe the constraint language we support. In Sections 6 and 7, I present the constraint normalization procedure. In Section 8, I present our experiments. In Section 9, I discuss related work. In Section 10, I conclude the chapter.

6.2 Motivation

6.2.1 Quantitative Information Flow for Side-Channel Analysis

Quantitative information flow is a powerful tool for understanding the capacity of side-channels to leak information. Quantitative information flow enables relaxing requirements of complete non-interference, allowing acceptably small side-channel leakage. Past work [23, 100] has augmented symbolic execution with model-counting to allow for the computation of the probability of program path. With additional static cost modeling of side-channel observables, these probabilities can be used to compute the mutual information between a side-channel

observable and the value of secret input. More recent work [25, 68, 69] has used quantification of side-channel leakage to synthesis side-channel attacks that allow an observer to learn the greatest amount of information. In both these lines of work, a key enabling technology is model-counting constraint solving.

6.2.2 String Constraints

The amount of string-manipulating code in modern software applications has been increasing. Common uses of string manipulation include: 1) Input sanitization and validation in web applications; 2) Query generation for back-end databases; 3) Generation of data formats such as XML and HTML; 4) Dynamic code generation; 5) Dynamic class loading and method invocation. In order to analyze programs that use string manipulation, it is necessary to develop techniques for efficient manipulation of string constraints. Recently, there has been significant amount of work in string constraint solving to address this problem [131–141]. One of our contributions in this paper is a constraint normalization and caching framework that can handle string constraints.

Consider the following string constraint F:

$$b = \text{“}https\text{”} \wedge \mathsf{prefix\_of}(b, url) \wedge c = \text{“?”} \wedge \mathsf{contains}(c, url)$$
$$\wedge\ w \in (0|1)^+ \wedge \mathsf{index\_of}(w, url) = 5$$

F is a conjunction of constraints on string variables b, c, w, and url. In addition to string equality and regular expression membership, string constraints such as prefix_of and contains and string functions such as index_of are commonly used in string manipulating code. The solution set of F is the set of string values that can be assigned to variables b, c, w, and url for which F evaluates to true.

Constraints such as F commonly arise in symbolic program analysis. For example, F can

correspond to a path constraint generated during symbolic execution of a string-manipulating program. A fundamental question about a constraint F generated during program analysis is its satisfiability. Symbolic program analysis techniques generate numerous satisfiability queries while analyzing programs. Given that satisfiability checking is computationally expensive, it is crucial to answer satisfiability queries efficiently in order to build scalable symbolic program analysis tools.

On the other hand, quantitative program analysis techniques ask another type of question while analyzing programs. Assume that we bound the length of the string variables b, c, w, and url in constraint F to 5. How many different string values are there for the variable b such that the constraint F is satisfiable within the given bound? This type of queries can be answered by model-counting constraint solvers. Again, due to the high complexity of model counting, answering model counting queries efficiently is crucial for quantitative program analysis.

Now, consider the following string constraint G:

$$k = \text{“\#”} \wedge w = \text{“}http:\text{”} \wedge \mathsf{contains}(k, var0) \wedge z \in (1|0)^{+}$$
$$\wedge\ \mathsf{index\_of}(z, var0) = 5 \wedge \mathsf{prefix\_of}(w, var0)$$

Constraint G is a constraint on string variables k, z, w, and $var0$. Assume that constraints F and G are generated during program analysis and it is necessary to check the number of satisfying solutions and satisfiability of each. Can we avoid making redundant calls to the constraint solver? Note that the solution sets of F and G are different since different string constants appear in these two constraints. However, the satisfiability and the cardinality of the solution sets for these two constraints are identical. Hence, if we were able to detect the relationship between the number of satisfying models of F and G and had stored the result of a model-counting query on F, then when we see G we do not have to call the model-counting constraint solver again. The same holds for satisfiability queries.

The question now becomes: Can we find a fast way to determine that F and G are equivalent with respect to satisfiability and model counting so that we can prevent redundant calls to the constraint solver? In this paper, we present a constraint normalization scheme to determine this type of equivalence. Based on our normalization scheme, the normalized form of F and G are identical:

$$v0 = \text{“}a\text{”} \wedge v1 = \text{“}bccde\text{”} \wedge \mathsf{contains}(v0, v2)$$
$$\wedge\ \mathsf{prefix\_of}(v1, v2) \wedge v3 \in (f|g)^+ \wedge \mathsf{index\_of}(v3, v2 = 5)$$

Hence, given two constraints, in order to determine their equivalence, we first normalize the constraints and check if their normalized forms are the same. Using a constraint store to cache the results of prior queries to the solver, we avoid redundant queries for constraints that have the same normalized form.

For both satisfiability and model-counting queries, we can cache the result of the query in a constraint store, use normalization to determine equivalence of constraints, and then reuse the query results from the store when we get a cache hit. However, since model-counting queries come with a bound parameter, in order for the query to match, the bound also has to match. If the bound does not match, can we still reuse a model counting query result? Parameterized model-counting techniques [125, 126] do not only count the number of solutions for a constraint within a given bound, but they also generate a model counter that can count the number of solutions for any given bound. Note that counting the number of solutions with different bounds may be necessary during program analysis. For example, consider the following constraint:

$$\mathsf{contains}(x, \text{“}abcde\text{”}) \wedge |y| > |x| \wedge y \in (ab)^*$$

This constraint has no solutions for bounds less than 5 but has satisfying solutions for higher bounds.

In this paper, we present a parameterized caching approach that utilizes parameterized

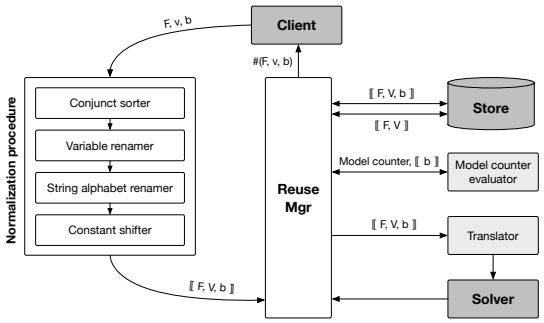


Figure 6.1: Architecture of Cashew

model-counting constraint solvers. We assume that, in response to a model-counting query, parameterized model-counting constraint solvers return a model-counting object that can be used to count the number of models for any given bound. By storing the model-counting object in the constraint cache store, we are able to reuse model counting query results even for queries with different bounds.

6.3 Constraint Caching

Our tool Cashew, depicted in Figure 6.1, is designed to work with a wide range of model-counting solvers to support quantitative program analyses. Algorithm 8 outlines how Cashew handles model-counting queries. Cashew expects a query of the form (F, V, b), where F is a well-formed formula, V is a set of variables in F, and b is a bound. The answer to the query, denoted as $\#(F, V, b)$, is the number of satisfying solutions for F for the variables in V within the bound b. We normalize the formula, variable(s) and bound using our normalization procedure, NORMALIZE-QUERY, which is described in the following sections. The resulting normalized query is denoted as $[\![F, V, b]\!] = \text{NORMALIZE-QUERY}(F, V, b)$.

Depending on the capabilities of the selected model-counting constraint solver, $[\![F, V, b]\!]$ is queried differently. Algorithm 8 outlines the normalization and query process. Typical model-counting constraint solvers [142–144], return a single count value ($\#(F, v, b)$) after receiving a

query of the form (F, V, b). For such constraint solvers, our caching algorithm first sends the query $[\![F, V, b]\!]$ to the cache store. If there is a cache hit, the result is returned to the client. If not, the normalized query is sent to the model-counting solver, and the result is stored under $[\![F, V, b]\!]$ and returned to the client.

We call a model-counting constraint solver *paramaterized* if it returns a model-counter object that can be used to compute the number of satisfying solutions for an arbitrary bound. ABC [126] is a parameterized model-counting constraint solver where the model-counter object is the transfer matrix of an automaton that accepts all satisfying models of the given constraint. SMC [125] and barvinok [145] are also parameterized model-counting constraint solvers where the model-counter object is a generating function.

For parameterized solvers, the store is queried as follows: First, $[\![F, V, b]\!]$ is queried. In the case of a hit, the result $(\#(F, V, b))$ is returned to the client. In the case of a miss, an additional query for $[\![F, V]\!]$ is made. If this results in a hit, the model counter for $[\![F, V]\!]$ is recovered from the store. This model counter is sent to the model-counter evaluator which evaluates $\#(F, V, b)$ based on $[\![b]\!]$. The result returned by the model-counter evaluator is stored under $[\![F, V, b]\!]$ and is returned to the client. If both queries are misses, a call to the selected solver is made, the model counter is computed and cached under the key $[\![F, V]\!]$, and $\#(F, V, b)$ is evaluated based on $[\![b]\!]$, stored under $[\![F, V, b]\!]$ and returned to the client.

In order to use Cashew's parameterized caching functionality and reuse cached model counters, a service that is able to take a model counter (such as a transfer matrix or a generating function) and evaluate it for a particular bound is required. This service is referred to as the model-counter evaluator.

Input: A query (F, V, b).
Output: The number of satisfying solutions of V in F under the length bound b.

$[\![F, V, b]\!]$ = NORMALIZE-QUERY(F, V, b) **if** *Hit on* $[\![F, V, b]\!]$ **then**
  return #(F,V,b)
end
if *Hit on* $[\![F, V]\!]$ **then**
  Evaluate the model counter for bound $[\![b]\!]$ using the model-counter evaluator;
  Store the result under $[\![F, V, b]\!]$
  return #(F,v,b)
end
Translate $[\![F, V, b]\!]$ and **send** it to the selected model-counting solver
Store the returned model counter under $[\![F, V]\!]$
Store #(F,V,b) under $[\![F, V, b]\!]$
return #(F,V,b)

Algorithm 8: CONSTRAINT-CACHING(F, V, b):

6.4 Group-Theoretic Framework

The goal of normalization is to reduce constraints equivalent under some property to the same form. This objective is shared by work in constaint programming, where detecting symmetries in constraints leads to a more efficient search [146, 147]. Symmetry-breaking for constraint programming is based on concepts from group theory [148] and we adopt this formulation to describe our normalization scheme for caching.

Our framework provides a means for constructing normal forms of constraints based on groups of property-preserving transformations. For different analysis problems, it might be necessary to preserve the entire solution set, the cardinality of the solution set, or only the satisfiability of constraints, each corresponding to a different level of normalization. Our framework is equally applicable, regardless of the desired level of normalization. Our framework is also not restricted to a constraint language, but is equally applicable to any background theory on which a group of property-preserving transformations can be defined.

Symmetry Groups

A group $(\mathcal{G}, \text{op})$ is a set of elements together with a binary operator that satisfies the four group axioms: closure, associativity, identity, and invertibility. For example, the set of all transpostions on the natural numbers, $\mathbb{N}$, under the binary operator function composition form a group. The transposition from $\mathbb{N}$ to itself defined by the relation $\{(1,2),(2,1)\}$ is an example of an element of this group which maps 1 to 2, 2 to 1 and all other elements of $\mathbb{N}$ to themselves.

A subset of a group is called a *subgroup* if it also forms a group under the same binary operator. We construct the group of cardinality-preserving transformations under composition $(\mathcal{G}_{\text{card}}, \circ)$ by introducing its generating subgroups. As composition is the only binary operator we consider, we simply refer to this group as $\mathcal{G}_{\text{card}}$ throughout the remainder of the paper.

Solution-Set-Preserving Subgroups of $\mathcal{G}_{\text{card}}$

A solution-set-preserving transformation is one under which the solution set is mapped to itself. Any solution-set-preserving transformation is trivially cardinality-preserving. Our first solution-set-preserving subgroup of $\mathcal{G}_{\text{card}}$ is the group mentioned above—that of all transpositions on $\mathbb{N}$. This group is identical to the permutation group on $\mathbb{N}$ whose elements fix all but a finite number of numbers. Because the indices of conjuncts within a constraint are natural numbers, the above subgroup can be said to act on the domain of indices $\mathcal{I}$.

Intuitively, this subgroup captures our understanding that the solution set of a constraint is independent of the order of the conjuncts in it. Under the transposition $\{(1,2),(2,1)\}$ for example, the formula $x > 0 \wedge y < 0$ is mapped to $y < 0 \wedge x > 0$, making the two orderings equivalent modulo the action of this group.

Our second solution-set-preserving subgroup is given by the transposition group on the infinite domain of possible variable names, $\mathbb{V}$. Since the solution set of a constraint is independent of the choice of variable names, two constraints that are equivalent modulo the action of this

group have the same solution set. As a simple example, realize that the number of solutions for $x < 7 \wedge x > 2$ is the same as that for $w < 7 \wedge w > 2$. This follows from the transposition given by the relation $\{(x, w), (w, x)\}$ being an element of this transposition group.

Cardinality-Preserving Subgroups of $\mathcal{G}_{\mathbf{card}}$

Preserving only the *cardinality* of the solution set of a constraint allows for the use of subgroups with more flexible group actions. Under these groups, the solution set of a constraint is bijectively mapped to the solution set of another constraint. The number of solutions remains unchanged under this bijection though the solution set itself is transformed.

Our first family of cardinality-preserving subgroups are given by the Euclidean groups $E(n)$ (symmetry groups on Euclidean space) acting on the solution space of linear integer arithmetic constraints. The elements of these groups are Euclidean motions including rigid motions such as translations and rotations as well as indirect isometries such as reflection. Under these symmetries of Euclidean space, the volume captured by the correspdoning polytope remains unchanged.

Though this volume is preserved under any action of the Euclidean group, the number of lattice points in the polytope can change under certain group actions. Because we are often interested in the number of integer solutions to a constraint, we limit ourselves to considering only those tranformations that preserve the number of lattice points as well as those that can be easily reflected through changes in the syntax of the constraint. In particular, our normalization scheme uses the subgroup of integral translations in Euclidean space as a generating subgroup for $\mathcal{G}_{\text{card}}$. Integral translations can be reflected syntacically in integer constraints through changes in the constant terms of each conjunct. Each constant term must be identically shifted by an integral amount. For example, shifting each constant term of the constraint $y = 2 \wedge x \geq 0 \wedge y \geq 0$ by 2 results in the constraint $x + y = 4 \wedge x \geq 2 \wedge y \geq 2$ which has the same number of integer solutions (6) as the original.

For any arithmetic constraint, F, we call the vector composed of the constant terms of each of its conjuncts its *shift*, denoted $\vec{Shift}(F)$ The domain that the subgroup of integral translations acts on is that of all possible shifts, which we denote $\mathcal{SH}$. For string or mixed constraints, we do not apply transformations from this subgroup and we say that $\vec{Shift}(F)$ of such constraints is $\emptyset$.

Our second cardinality-preserving subgroup is given by the group of permutations on the string alphabet, Σ. The solution set of a string constraint can be canonically represented by an automaton that accepts exactly the set of solutions to that constraint. Transitions between states are made based on a set of allowed alphabet symbols. Permuting the set of alphabet symbols thus changes the strings accepted by that automaton but not the cardinality of the accepted set. As a simple example, observe that the number of solutions for the constraints $F := x.\mathsf{contains}(\text{“}ac\text{”})$ and $F' := x.\mathsf{contains}(\text{“}bd\text{”})$ are the same.

Orbits under the Symmetry Group $\mathcal{G}_{\text{card}}$

These subgroups generate $\mathcal{G}_{\text{card}}$ in the following sense: the domain of any element of $\mathcal{G}_{\text{card}}$ is the union of the domains of the subgroups, making it the Cartesian product $\mathcal{I} \times \mathbb{V} \times \mathcal{SH} \times \Sigma$. Every element of a subgroup acts as an element of $\mathcal{G}_{\text{card}}$ by acting as the identity on every domain element on which it is not defined. Any element of $\mathcal{G}_{\text{card}}$ can be written as a composition of elements from these subgroups.

For a constraint F, the **orbit** of F under $\mathcal{G}_{\text{card}}$ is the set of constraints obtained by applying any element $\sigma \in \mathcal{G}_{\text{card}}$ to F.

The problem of choosing a normalized form for F can now be formulated as choosing a representative constraint from the orbit of F under $\mathcal{G}_{\text{card}}$. We do this by defining a strict ordering on constraints and choosing the well-defined lowest ordered constraint within the orbit as the representative for all constraints within the orbit.

While we have spoken generally about cardinality-preserving group actions, our application

$$
\begin{aligned}
\mathcal{T}_{\mathsf{S}} &:= \mathsf{c} \mid \mathsf{v}_{\mathsf{S}} \mid \mathcal{T}_{\mathsf{S}} \cdot \mathcal{T}_{\mathsf{S}} \mid \mathsf{char\_at}(\mathcal{T}_{\mathsf{S}}, \mathcal{T}_{\mathsf{A}}) \mid \mathsf{int\_to\_str}(\mathcal{T}_{\mathsf{A}}) \mid \\
&\quad \mathsf{replace}(\mathcal{T}_{\mathsf{S}}, \mathcal{T}_{\mathsf{S}}, \mathcal{T}_{\mathsf{S}}) \mid \mathsf{substr}(\mathcal{T}_{\mathsf{S}}, \mathcal{T}_{\mathsf{A}}, \mathcal{T}_{\mathsf{A}}) \\
\mathcal{T}_{\mathsf{R}} &:= \epsilon \mid \mathsf{s} \mid (\mathcal{T}_{\mathsf{R}}) \mid \mathcal{T}_{\mathsf{R}} \cdot \mathcal{T}_{\mathsf{R}} \mid \mathcal{T}_{\mathsf{R}} | \mathcal{T}_{\mathsf{R}} \mid \mathcal{T}_{\mathsf{R}}^{*} \\
\mathcal{T}_{\mathsf{A}} &:= \mathsf{n} \mid \mathsf{v}_{\mathsf{A}} \mid -\mathcal{T}_{\mathsf{A}} \mid (\mathcal{T}_{\mathsf{A}}) \mid \mathcal{T}_{\mathsf{A}} + \mathcal{T}_{\mathsf{A}} \mid \mathcal{T}_{\mathsf{A}} - \mathcal{T}_{\mathsf{A}} \mid \mathcal{T}_{\mathsf{A}} \times \mathsf{n} \mid \\
&\quad \mathsf{n} \times \mathcal{T}_{\mathsf{A}} \mid |\mathcal{T}_{\mathsf{S}}| \mid \mathsf{index\_of}(\mathcal{T}_{\mathsf{S}}, \mathcal{T}_{\mathsf{S}}) \mid \mathsf{str\_to\_int}(\mathcal{T}_{\mathsf{S}})
\end{aligned}
$$

Figure 6.2: Here $\mathsf{c} \in \Sigma$, $\mathsf{n} \in \mathbb{Z}$, $\mathsf{s} \in \Sigma^*$, v_{S} and v_{A} denote an unbounded string variable and an integer variable, resp.

of interest is in *parameterized* model-counting which involves finding the number of satisfying solutions to a constraint for any given bound. While most of the group actions defined above preserve the number of solutions for a given bound, the elements of the Euclidean group may not. For example, $y = 2 \wedge x \geq 0 \wedge y \geq 0$ has 6 solutions given a bound of 2 but $x+y = 4 \wedge x \geq 2 \wedge y \geq 2$, which is in the same orbit under $\mathcal{G}_{\text{card}}$, has only one solution for the same bound. In order to preserve the paramaterized model count, the bound is translated according to the same group action as the constraint. In the example above, bound 2 is translated to bound 4 by the same integer translation (2) that translated the shift, resulting in 6 satisfying models.

6.5 Constraint Language

We focus on constraints over *strings* and *linear integer arithmetic*.

We define three types of terms: string terms $\mathcal{T}_{\mathsf{S}}$, regular expression terms $\mathcal{T}_{\mathsf{R}}$, and linear integer arithmetic terms $\mathcal{T}_{\mathsf{A}}$, as described in Figure 6.2. We consider three types of constraints over these terms, which we call *conjuncts* throughout this paper: string conjuncts S, regular membership conjuncts R, and LIA conjuncts A. Let L be a language defined over these conjuncts, formalized in Figure 6.3. Input constraints to our normalization procedure are assumed to be in conjunctive form, where each conjunct is from L.

The set of string operators S_{op} is comprised of the operators listed in the definition of $\mathcal{T}_{\mathsf{S}}$ and the set of the string comparators S_{comp} by the comparator listed in S; the set of regular

$$
\begin{aligned}
\mathsf{L} &\coloneqq \top \mid \bot \mid \mathsf{S} \mid \mathsf{R} \mid \mathsf{A} \\
\mathsf{S} &\coloneqq \mathcal{T}_\mathsf{S} = \mathcal{T}_\mathsf{S} \mid \mathcal{T}_\mathsf{S} \neq \mathcal{T}_\mathsf{S} \mid \mathsf{contains}(\mathcal{T}_\mathsf{S}, \mathcal{T}_\mathsf{S}) \mid \mathsf{prefix\_of}(\mathcal{T}_\mathsf{S}, \mathcal{T}_\mathsf{S}) \mid \\
&\quad \mathsf{suffix\_of}(\mathcal{T}_\mathsf{S}, \mathcal{T}_\mathsf{S}) \mid \mathsf{not\_contains}(\mathcal{T}_\mathsf{S}, \mathcal{T}_\mathsf{S}) \mid \\
&\quad \mathsf{not\_prefix\_of}(\mathcal{T}_\mathsf{S}, \mathcal{T}_\mathsf{S}) \mid \mathsf{not\_suffix\_of}(\mathcal{T}_\mathsf{S}, \mathcal{T}_\mathsf{S}) \\
\mathsf{R} &\coloneqq \mathcal{T}_\mathsf{S} \in \mathcal{T}_\mathsf{R} \mid \mathcal{T}_\mathsf{S} \notin \mathcal{T}_\mathsf{R} \\
\mathsf{A} &\coloneqq \mathcal{T}_\mathsf{A} = \mathcal{T}_\mathsf{A} \mid \mathcal{T}_\mathsf{A} < \mathcal{T}_\mathsf{A} \mid \mathcal{T}_\mathsf{A} \leq \mathcal{T}_\mathsf{A} \mid \mathcal{T}_\mathsf{A} \neq \mathcal{T}_\mathsf{A}
\end{aligned}
$$

Figure 6.3: The language L: conjuncts of string (S), regular expression (R) and linear integer arithmetic (A) types.

expression operators R_{op} by those in $\mathcal{T}_\mathsf{R}$ and the set of regular expression comparators R_{comp} by those listed in R; the set of the linear integer arithmetic operators A_{op} by those in $\mathcal{T}_\mathsf{A}$ and the set of comparators A_{comp} by those listed in A. By $\textsc{Type}(\_)$ we denote a function that takes in a conjunct and returns the comparator of this conjunct. For every term $t \in \mathcal{T}_\mathsf{S} \cup \mathcal{T}_\mathsf{R} \cup \mathcal{T}_\mathsf{A}$, and conjunct $C \in \mathsf{L}$, we define a *length function* denoted as $\| t \|$, respectively $\| C \|$, as a total number of variables, constant symbols, and operators in t, respectively in C. The expression $\#Var(t)$ returns the number of variables in t.

6.6 Constraint Ordering

Assume a strict total ordering on constraints, $\prec$. A constraint F is a normal form if for every other constraint F' in its orbit under $\mathcal{G}_{\text{card}}$, $F \prec F'$. There are many ways to impose an ordering on constraints. We present one possible ordering below.

Our ordering produced compositionally, with strict orders defined over various components of our language which are composed to yield an ordering on constraints. To start, we define an ordering on each element of the domain of $\mathcal{G}_{\text{card}}$.

The ordering on both $\mathbb{V}$ and Σ is lexicographical. The ordering on $\mathcal{I}$ is that induced by the natural numbers. We define the ordering on $\mathcal{SH}$, the domain of constant shifts, after we introduce an ordering on vectors. We consider vectors over strict totally ordered sets and denote by $\prec_{vec}$ an order on such vectors.

Let X be a strict totally ordered set, and $\prec_X$ be a strict total order on X. Let $\boldsymbol{v} =$

$(v_0, \ldots, v_n)$ and $\boldsymbol{u} = (u_0, \ldots, u_m)$ be two vectors over X, then $\prec_{vec}$ is defined as:

$$\boldsymbol{v} \prec_{vec} \boldsymbol{u} \iff \begin{cases} m < n, \textit{ or} \\ m = n, \exists i \forall j : \ j < i < n, v_j = u_j, v_i \prec_X u_i. \end{cases}$$

This defines ordering on $\mathcal{SH}$ since shift vectors are built over integer constants.

Our normalization procedure relies on the following auxiliary functions that given a constraint return, as vectors, various structural and syntactic components characterizing the constraint. These vectors are built over the domains of $\mathbb{V}$, Σ and $\mathbb{Z}$, i.e. over strict totally ordered sets.

$\vec{VI}(F)$ — returns *a vector of the indices of variables* as they occur in F relative to other variables, constants and operators. The indices are compared according to the "$<$" operator over $\mathbb{Z}$.

$\vec{Int}(F)$ — returns *a vector of integer constants* occuring in F from left to right, ignoring all elements of $\vec{Shift}(F)$. The vectors are compared according to the "$<$" operator on $\mathbb{Z}$.

$\mathbb{V}(F)$ — returns *a vector of variable names* occuring in F from left to right. These vectors are compared according to the lexicographical order on $\mathbb{V}$.

$\boldsymbol{\Sigma}(F)$ — returns *a vector of string characters* occuring in F from left to right. The vectors are compared according to the lexicographical order on Σ.

Next, we define strict total orderings on operators and (separately on) comparators, listing them in the order of the increasing precedence. Both, operators and comperators, are ordered with precedence to S, then R, and A.

S_{op}: $\cdot$, the rest of the string operators in the lexicographic order according to their names in Figure 6.2;

R_{op}: ordered according to the standard precedence order on regular expression operators;

A_{op}: $+$,$-$, $\times$, $||$, (), the rest of the LIA operators in the lexicographic order according to their names in Figure 6.2.

S_{comp}: $=$, $\neq$, the rest of the string comperators in the lexicographic order according to their names in Figure 6.3;

R_{comp}: $\in$, $\notin$;

A_{comp}: $=$, $<$, $\leq$, $\neq$;

The ordering on comparators allows to define an order $\prec_{type}$ on the types of the conjuncts Type, based on the type of the comparator occuring in the conjuncts. The strict total ordering on operators allows to introduce *vectors of operators* of constraints and compare them with $\prec_{vec}$:

$\vec{Op}(F)$ — *a vector of string, regular and LIA operators* occuring in F from left to right.

Note, all auxiliary vectors, and their orderings, introduced in this section are defined for constraints and are naturally applicable to conjuncts – as to a special type of constraints with a single conjunct.

In the future, when we compare two elements of the same type we will drop the subscript notation and use $\prec$ to represent comparison between them.

We are now ready to build a strict total order on conjuncts. We define the ordering hierarchically: the structural or syntacic aspects of the conjuncts are compared one at a time in a fixed order, until a tie-breaking aspect is found. This order can be selected in any way. We present one intuitive order below to distinguish conjuncts with more significant differences as early as possible. The conjuncts are first compared based on their type Type, then based on their length $\|\,\|$, then the total number of variables $\#Var$, then their vectors of operators $\vec{Op}$, followed by the vectors of indices of variables $\vec{VI}$, their vectors of integer coefficients $\vec{Int}$,

their vectors of variable names $\mathbf{V}$, then vectors of string constants $\mathbf{\Sigma}$, and finally based on their constant shifts $\vec{Shift}$. This order is described in Algorithm 9.

Require : Two conjuncts $C_1, C_2 \in \mathsf{L}$
Ensure : TRUE if $C_1 \prec C_2$, otherwise FALSE
for $f \in [\text{TYPE}, \|\ \|, \#Var, \vec{Op}, \vec{VI}, \vec{Int}, \mathbf{V}, \mathbf{\Sigma}, \vec{Shift}]$ **do**
  if $f(C_1) \prec f(C_2)$ **then**
    return TRUE
  end
end
return FALSE

Algorithm 9: C-LESSTHAN(C_1, C_2): Conjunct Comparison

This order is strict and total. Two conjuncts are equal if and only if they are the same conjunct. This allows us to extand the ordering to constraints as follows:

(i) Order constraints based on their total number of conjuncts.

(ii) Then, order constraints by comparing their conjuncts element-wise in order of precedence according to the order imposed on $\mathcal{I}$. This is equivalent to comparing conjuncts pairwise from first to last.

(iii) A constraint F comes before a constraint G if the first differing conjunct of F is lower ordered than that of G under C-LESSTHAN.

This order is described in Algorithm 10.

Require : Two constraints $F = F_1 \wedge \ldots \wedge F_m$ and $G = G_1 \wedge \ldots \wedge G_n$
Ensure : TRUE if $F \prec G$, otherwise FALSE
if $m = n$ **then**
  for $i \leftarrow 1, n$ **do**
    if ***C-LessThan***(F_i, G_i) **then**
      return TRUE
    end
  end
end
return $m < n$

Algorithm 10: F-LESSTHAN(F, G): Constraint Comparison

6.7 Constraint Normalization Procedure

The normal form of a constraint F is the lowest constraint in the orbit of F under $\mathcal{G}_{\text{card}}$. In this section, we present a normalization procedure to find the normal form of a constraint.

Given a transformation $\sigma \in \mathcal{G}_{\text{card}}$, we define $\sigma[F]$, the *action of* σ *on* F, as a composition of elements of four categories corresponding to each of the components of the domain of $\mathcal{G}_{\text{card}}$:

$\mathcal{I}$: $\sigma_{\mathcal{I}}[F]$ gives the constraint resulting from re-ordering the conjuncts of F according to $\sigma_{\mathcal{I}}$.

$\mathbb{V}$: $\sigma_{\mathbb{V}}[F]$ gives the constraint resulting from renaming the variables of F according to $\sigma_{\mathbb{V}}$;

Σ: σ_{Σ}[F] gives the constraint resulting from permuting the alphabet constants in F according to σ_{Σ};

$\mathcal{SH}$: $\sigma_{\mathcal{SH}}[F]$ gives the constraint resulting from shifting each element of F's shift according to $\sigma_{\mathcal{SH}}$.

We first present an expensive but complete procedure for normalization in Algorithm 4 and give guarantees for its termination and correctness. Given a constraint F, this procedure probes each permutation F' of conjuncts in F, building and applying a composite σ from transformations specific to the domains $\mathbb{V}$, Σ, and $\mathcal{SH}$ which reduces F' until the only transformations that can reduce it further involve an action on $\mathcal{I}$. The results among all permutations of F are compared and the lowest-ordered result is chosen as the normal form of F.

The procedure uses auxiliary functions to build the minimizing domain-specific transformations:

MIN-σ-$\mathbb{V}(F')$ constructs $\sigma_{\mathbb{V}}$ compositionally — it proceeds through the conjuncts of F' from left to right renaming the variables of F' in order of appearance. Each time a new variable is encountered a transposition is added to the composition that permutes the name of the encountered variable and the lowest-ordered variable name that no other variable of F' has been renamed to yet. At the start of the procedure, $\sigma_{\mathbb{V}}$ is initialized to the identity transposition on $\mathbb{V}$.

MIN-σ-$\Sigma(F')$ similarly constructs σ_{Σ} — it proceeds through the conjuncts of F' from left to right, this time permuting string characters. Each time a new string character is encountered, a transposition is added to the composition that permutes the encountered string character with the lowest-ordered character that no other character in F' has been mapped to yet. σ_{Σ} is initialized as the identity transposition on Σ.

MIN-σ-$\mathcal{SH}(F')$ returns $\sigma_{\mathcal{SH}}$ — the tranformation on $\vec{Shift}(F)$ that translates the constant coefficient of the first appearing (from left to right) linear integer arithmetic conjunct in F' to 0. If F contains variables that are shared between string and LIA constraints $\sigma_{\mathcal{SH}}$ is the identity transformation.

Input: A constraint F
Output: The normalized form of F

$F_{min} := F$ **foreach** *permutation* F' *of conjuncts in* F **do**
  $\sigma_{\mathbb{V}} :=$ MIN-σ-$\mathbb{V}(F')$
  $\sigma_{\Sigma} :=$ MIN-σ-$\Sigma(F')$
  $\sigma_{\mathcal{SH}} :=$ MIN-σ-$\mathcal{SH}(F')$
  $F' := \sigma_{\mathbb{V}} \circ \sigma_{\Sigma} \circ \sigma_{\mathcal{SH}}[F']$
  if ***F-LessThan****(F', F_{min})* **then**
    $F_{min} := F'$
  end
end
return F_{min}

Algorithm 11: COMPLETE-NORMALIZATION(F)

Theorem 1 *Algorithm 11 terminates.* *Proof: Given a constraint F, there are finitely many permutations of conjuncts F'. Consequently, there are finitely many executions of the "for*

each" loop. Construction of each permutation F' is linear in the length of F. Construction of each of the domain-specific transformations within a single "for each" call is performed in a single pass through the conjuncts of F', thus, is linear in the length of F, too. Computation of the action of the final transformation on F' is also linear in the length of F. Thus, Complete-Normalization *terminates.*

■

Theorem 2 *Algorithm 11 returns the normal form of F.*

Proof: Assume $G =$ Complete-Normalization(F) *is not the normal form of F. Then either G is not in the orbit of F under $\mathcal{G}_{card}$ or there is some constraint H in the orbit of F such that $H \neq F$ and $F \not\prec H$. We show that both result in a contradiction.*

Assume G is not in the orbit of F. G is the result of permuting the conjucts of F, the action of some $\sigma_{\mathcal{I}}$, composed with domain specific transformations. Each domain-specific transformation has an inverse in $\mathcal{G}_{card}$ as does any permutation of the conjuncts of F. Therefore, there exists some σ in $\mathcal{G}_{card}$ such that $\sigma[G] = F$, meaning that G and F are in the same orbit.

Now assume that there is some H in the orbit of F such that $H \neq G$ and $G \not\prec H$. The order of conjuncts in H is given by some transposition of the indices of F. This means that there is some iteration of the for loop of Algorithm 4 in which the conjuncts of the considered permutation of F are ordered identically to those of H. By construction, our choices of $\sigma_{\mathbb{V}}$, σ_{Σ} and $\sigma_{\mathcal{SH}}$ reduce this constraint to the lowest-ordered constraint that maintains the same ordering of conjuncts. Therefore either $G = H$ or $G \prec H$.

■

Algorithm 11 gives a normalization procedure which is *sound* (each orbit has at least one fixed point) and *complete* (there is exactly one fixed point for each orbit). In practice, however, such a brute force exploration is very expensive. For our implementation, we use a sound but not complete normalization procedure given in Algorithm 5. Given F, Normalization(F)

returns the semi-normal form on F — a constraint within the orbit of F which, though not necessarily the lowest in the orbit, is not higher ordered than F.

Algorithm 12 simplifies COMPLETE-NORMALIZATION procedure in that instead of brute-forcing all permutations of conjuncts in F, it inexpensively chooses a permutation by ordering the conjuncts of F according to C-LESSTHAN up to the point when further refinement involves comparison over the domains $\mathbb{V}$, Σ, or $\mathcal{SH}$. In other words, the conjuncts are not compared according to their variable names, string constants or shifts. It is possible that two conjuncts in F are equal by this comparison, in which case their initial order in F is preserved. The resulting permutation of conjuncts defines a transposition on $\mathcal{I}$. We apply this transposition to F, resulting in a constraint F'. $\sigma_{\mathbb{V}}$, σ_{Σ}, and $\sigma_{\mathcal{SH}}$ are generated by the same auxiliary functions as in Algorithm 4, composed, and applied to F'. The result is the semi-normal form of F.

Input: A constraint F
Output: A semi-normal form of F

$F' :=$ **Permute** conjuncts of F according to Algorithm 9 up until $\mathbf{V}$
$\sigma_{\mathbb{V}} :=$ MIN-σ-$\mathbb{V}(F')$
$\sigma_{\Sigma} :=$ MIN-σ-$\Sigma(F')$
$\sigma_{\mathcal{SH}} :=$ MIN-σ-$\mathcal{SH}(F')$
$[\![F]\!] := \sigma_{\mathbb{V}} \circ \sigma_{\Sigma} \circ \sigma_{\mathcal{SH}}[F']$
return $[\![F]\!]$

Algorithm 12: NORMALIZATION(F)

Theorem 3 *Algorithm 12 is sound. Proof: Each action on F is the action of an element of $\mathcal{G}_{card}$. By definition, the resulting formula is in the orbit of F under $\mathcal{G}_{card}$.* ■

The procedure given in Algorithm 12 is not complete. There are orbits for which not every constraint is reduced to the same form. Though this potentially increases the number of misses to the cache, our experimental results demonstrate the large number of formulas mapped to the same semi-normal form by Algorithm 5.

Queries to Cashew are of the form (F, V, b) where V is the set of variables on which to count, and b is the maximum length of a satisfying solution. To ensure that the cardinality of the solution set is preserved after normalizing F, both V and b must be normalized according to the

same transformations applied to F during Algorithm 13. Procedure NORMALIZE-QUERY(F, V, b) implements the query normalization.

Input: A query (F, V, b)
Output: A normalized query $[\![F, V, b]\!]$

$[\![F]\!] :=$ NORMALIZATION(F)
$\sigma :=$ the transformation used to normalize F
$[\![V]\!] := \sigma[V]$
$[\![b]\!] := \sigma[b]$
return $([\![F]\!], [\![V]\!], [\![b]\!])$

Algorithm 13: NORMALIZE-QUERY(F, V, b)

6.8 Experimental evaluation

We implemented our tool, Cashew, as an extension of the Green [127] caching framework. This allows Cashew to use any of the existing Green services, and it allows Green users to benefit from our normalization procedure. We experiment with Cashew-enabled satisfiability and model-counting services, which support string constraints and linear integer arithmetic. They also support *mixed* constraints, i.e., those involving both string and arithmetic operations. In this evaluation, we used ABC [126] as our constraint solver. As we explained in Section 6.3, other model-counting constraint solvers can be integrated instead of ABC by providing an appropriate translator (and, optionally, a model-counting-object evaluator).

All the experiments were run on an Intel Core i7-6850 3.5 GHz computer running Linux 4.4.0. The machine has 128 GB RAM, of which 4 GB were allocated for the Java VM.

6.8.1 Model counting over the SMC/Kaluza string constraint dataset

The Kaluza dataset is a well-known benchmark of string constraints that are generated by dynamic symbolic execution of real-world JavaScript applications [134]. The authors of the SMC solver [125] translated the satisfiable constraints to their input format: one contains 1,342

big, while the other contains 17,554 small where big and small classification is done based on the constraint sizes in the Kaluza dataset. We shall refer to the former as the original SMC-Big and to the latter as the original SMC-Small.

Duplicate constraints

While inspecting the results of our normalization, we found out that many of the files within each dataset are identical (indistinguishable by `diff`). Due to the presence of duplicates, even trivial caching (without any normalization) will yield some benefit on the original datasets. After removing all duplicate files, only 359 of the 1,342 constraints in SMC-Big and 9,745 of the 17,554 constraints in SMC-Small were found to be unique. As we discuss below, our normalization procedure allows further reductions in this dataset, increasing the benefits of caching well beyond what can be achieved with trivial caching.

Model counting

Since these constraints correspond to path conditions from symbolic execution, counting the number of satisfying models of each one could be necessary for quantitative analysis. We model-counted all constraints in each set as a simple way to emulate the behavioral pattern (w.r.t. caching) of one or more users performing quantitative analyses on the original programs.

When counting the models of a constraint over unbounded strings, in order to avoid infinite counts, one needs to set a bound on the length of strings to be considered. In this experiment, we set the bound to 50 characters for both sets. We ran the whole dataset (model-counting each constraint) first without normalization or caching, and then again with Cashew normalization and caching enabled. In non-caching mode, each constraint was sent unmodified to the model-counting solver. In caching mode, the cache was cleared before running SMC-Big, and again before running SMC-Small. Since these path constraints were produced by an external symbolic executor, in this experiment we did not use SPF. Instead, Cashew was run in standalone mode.

Note that since all constraints were model-counted, the order in which we traverse the datasets does not matter, as each normalized constraint will fall within some orbit, and for that orbit, the full cost of model counting will be paid exactly once (on the first cache miss).

Results

Table 6.1 shows the total, maximum and average model-counting time, as well as the speedups obtained by Cashew on each of these metrics, for the two datasets with and without duplicates. On the SMC-Big set, Cashew achieved a speedup over 10x. On the SMC-Small set, which is a rather bad case for the caching trade-off because it contains a large number of very small constraints, Cashew still achieved a 2.19x speedup.

For the original datasets, these numbers (e.g., a 87x speedup) are largely due to the presence of duplicates, which makes even trivial caching very effective. We report the results because the original datasets are widely used, and because the duplicates might indeed have been genuinely generated by symbolic execution of various different (yet similar) JavaScript programs.

Figure 6.4 depicts the effect of our normalization procedure on the original benchmarks. The area of each orbit is proportional to its size. Labels indicating orbit size are shown only when they fit in the available space. For the original SMC-Small set, the 17,554 original constraints are reduced to 360 orbits. For the SMC-Big set, the 1,342 original constraints are reduced to just 34 orbits.

We do not compare Cashew with Green because the original Green (without Cashew) cannot handle string constraints.

Note that the largest constraint in SMC-Small takes slightly more time after normalization. We cannot infer much from this, because the largest constraint in SMC-Small barely takes one second; the small difference (about 30 msec) could be due to noise. However, the maximum time for SMC-Big decreased by 3x with caching enabled, from 122 to 40 seconds. This is due to normalization. The constraint that (without normalization) requires maximum time to be

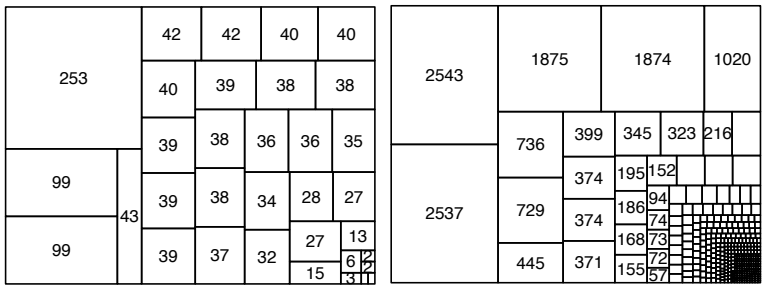


Figure 6.4: Orbit sizes for the original SMC datasets.

		Without caching	With caching	Speedup
Big (no dups)	Average	8.94 s	0.82 s	10.90x
	Maximum	121.92 s	40.13 s	3.03x
	Total time	3,208.65 s	293.21 s	10.94x
Small (no dups)	Average	0.12 s	0.05 s	2.40x
	Maximum	1.09 s	1.12 s	0.97x
	Total time	1,211.09 s	552.56 s	2.19x
Big (original)	Average	23.32 s	0.26 s	89.70x
	Maximum	121.92 s	40.13 s	3.03x
	Total time	31,297.90 s	358.17 s	87.38x
Small (original)	Average	0.13 s	0.05 s	2.60x
	Maximum	1.09 s	1.12 s	0.97x
	Total time	2,221.91 s	971.50 s	2.29x

Table 6.1: Model counting SMC-Big and SMC-Small.

model-counted falls within some orbit. It does not matter which constraint in that orbit will be the one to cause a cache miss once caching is enabled — only one of them will, and as they are all normalized to the same normal form, any of them would take the same model-counting time. What is interesting is that said time can be significantly smaller than the maximum pre-normalization time.

Table 6.2 shows the number of orbits that are achieved by different subsets of the transformations in our normalization procedure. Since some transformations can benefit from others,

Transformations enabled	#Orbits (SMC-Big)	#Orbits (SMC-Small)
None	359	9754
All transformations	34	360
All except $\sigma_{\mathcal{I}}$	72	376
All except $\sigma_{\mathbb{V}}$	344	9645
All except σ_{Σ}	35	841
All except removeVar	34	361
All except removeConj	40	386

Table 6.2: Effect of transformations on orbit refinement.

instead of considering them in isolation, we measured the effect of disabling each one. We did not include $\sigma_{\mathcal{SH}}$ as it doesn't apply to the string domain. The removeVar and removeConj transformations are preprocessing steps that remove redundant variables and conjuncts, respectively. These results indicate that all transformations yield some benefit, and that $\sigma_{\mathbb{V}}$ is the most beneficial transformation overall. For SMC-Small, removing σ_{Σ} more than doubles the number of orbits. The same is true of $\sigma_{\mathcal{I}}$ for SMC-Small. This shows that different transformations can be more effective for different datasets.

6.8.2 SPF analysis of string-handling code

In this second part of the experimental evaluation we use Symbolic PathFinder [130] with Cashew, to symbolically execute Java programs that operate on strings. In order to support model-counting-based quantitative analyses, we are interested in obtaining a model count for each leaf path constraint.

As an example of quantitative information flow analysis, we study some possible applications of Cashew to side-channel analysis. We consider four Java programs in which a side channel can allow an attacker to gain information about a hidden secret. `PasswordCheck1` contains a method that checks whether or not a user-given string matches a secret password. Due to the way the program is written, the attacker can deduce that the longer the program executes, the longer a prefix of the hidden password was matched. `PasswordCheck2` is another variant that attempts to mitigate that vulnerability by requiring a certain number of characters to be compared before returning, even if a mismatch has already been found. This yields a more interesting side channel, which can still be exploited but is much noisier and less predictable. `Obscure` is a Java translation of the `obscure.c` program used in [125], which is a password change authorizer. Given an old password and a new one, `Obscure` performs a series of tests to determine whether the new password is different enough from the old one. `CRIME` is a Java

version of a well-known attack, *Compression Ratio Info-leak Made Easy* [100, 149]. This is a side channel in space — the secret is concatenated with a string that can be controlled by the attacker, and both are compressed together before encryption. Thus, the attacker can try various strings and observe the changes in the size of the compressed payload to infer, from the compression rate, the level of similarity between the secret and the injected content.

In symbolic execution, it is not always desirable to make *all* arguments of a method symbolic. This is often the case due to scalability issues. It can also be due to the need to explore a nonstandard distribution of some parameter. Consider, for instance, a situation where a list of passwords from a website is unwillingly disclosed to the public. As a consequence, users are strongly encouraged to change their passwords, and an algorithm similar to the `Obscure` program is employed to ensure that they are sufficiently different from the stolen ones. We might be interested in measuring the amount of leakage of the algorithm over that particular list of passwords. By running SPF on `Obscure` with the new password as a symbolic string, and the old password as a concrete string, we can measure the leakage for that particular stolen password. By repeating this for various passwords from the list, we can quantify the algorithm's leakage for that list's particular distribution. Doing so requires running SPF repeatedly on the same code, but with different secret strings. This will affect many path conditions in fundamental ways, but others might be unaffected, or changed in such a way that Cashew can still normalize them down to a previously seen one.

RockYou

The RockYou1K dataset is a sample of 1,000 real-world passwords taken from the RockYou leak [150] without duplicates. The sample consists of 1,000 unique passwords that cover all lengths between 6 and 16 characters, and can include any ASCII symbols.

Program	Caching	Total time	Speedup	#Hits	#Misses	H/M
Password1	None	297 s	-	-	-	-
	Trivial	258 s	1.15x	17,547	56,173	0.31
	Cashew	106 s	2.80x	62,797	10,923	5.75
Password2	None	3,364 s	-	-	-	-
	Trivial	3,379 s	0.99x	30,448	824,832	0.04
	Cashew	1,243 s	2.71x	659,804	195,476	3.38
Obscure	None	2,158 s	-	-	-	-
	Trivial	1,965 s	1.10x	2,000	59,000	0.03
	Cashew	609 s	3.54x	44,893	16,107	2.79
CRIME	None	3,005 s	-	-	-	-
	Trivial	2,941 s	1.02x	31,884	84,127	0.38
	Cashew	1,067 s	2.82x	78,289	37,722	2.08

Table 6.3: SPF-based quantitative analyses of string programs.

Results

For each of the four programs under analysis, we ran 1,000 symbolic-execution-based side-channel analyses, using as the secret each of the 1,000 passwords in the RockYou1K dataset. For `PasswordCheck1` and `PasswordCheck2`, the secret is the password, which is concrete, and the user's guess is a symbolic string. For `Obscure`, the roles are reversed: what we made concrete is the old password, which is no longer secret, whereas the user-chosen new password (which is secret) is symbolic. For `CRIME`, we used a concrete secret (session ID) and a symbolic user-injected payload.

Table 6.3 shows execution time, hits and misses for three execution modes. The first mode uses neither normalization nor caching. In the second mode, only *trivial caching* is performed — normalization is disabled, which measures the extent to which syntactically identical path conditions (akin to the duplicates mentioned in Section 6.8.1) are generated. In the third mode, Cashew's normalization is enabled. Note that each symbolic execution generates many path conditions. The tables show the aggregated results over all path conditions of each execution and the 1,000 executions of each mode. As in the previous section, we do not compare Cashew with Green in these experiments because the original Green (without Cashew) cannot handle string constraints.

	Total time (no cache)	Total time (Cashew)	Total norm. time	Total norm. calls	Average norm. time	Total cache size
SMC-Big	3,209 s	293 s	<3 s	359	8 ms	54 KB
SMC-Small	1,211 s	553 s	31 s	9,745	3 ms	104 KB
Password1	297 s	106 s	20 s	73,720	275 μs	2.9 MB
Password2	3,364 s	1,243 s	94 s	855,280	110 μs	58 MB
Obscure	2,158 s	609 s	19 s	61,000	300 μs	5.1 MB
CRIME	3,005 s	1,067 s	47 s	116,011	400 μs	8.8 MB

Table 6.4: Cashew normalization and caching costs.

The results show that, for these experimental subjects, Cashew achieved an average speedup of nearly 3x, while trivial caching only achieved 1.06x on average (and, for `PasswordCheck2`, was in fact slower than no caching). Additionally, the hit/miss ratios improve dramatically when switching from trivial caching to Cashew.

Costs of caching and normalization

A caching scheme involves overheads and space/time trade-offs. Normalization overhead must be kept low, since its cost must be paid not only for each hit, but also for each miss. Cache size must also be kept within reasonable limits. Cashew is implemented on top of Green and, like Green, uses the in-memory Redis [151] database by default. This allows extremely fast queries, but competes with the client application for available RAM. As Table 6.4 shows, the average time to normalize a constraint in our SPF symbolic execution experiments was only a few hundred microseconds. It was about 8 milliseconds for the largest formulas (the SMC-Big set, with an average size before normalization of about 10 KB of text per constraint). Finally, as shown in Table 6.4, the total cache memory usage was very reasonable for these experiments.

6.8.3 Parameterized caching

In this last part of the experimental evaluation we present some experiments for evaluating parameterized caching—that is, caching that leverages parameterized model-counting solvers.

The motivation behind these experiments is that users of Cashew who are targeting quanti-

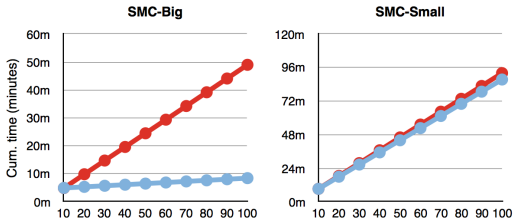


Figure 6.5: SMC datasets for increasing bounds. Cashew: non-parameterized (above) and parameterized (below) caching.

tative information analysis techniques often perform their analyses with various different bound values. For example, in side-channel analysis, one may want to compute the amount of information leakage for different lengths of a symbolic secret or input. This requires using different bounds when counting models. In scenarios where we have reason to believe that there is potential for reusing already-created model-counting objects for multiple values of b, we can try to amortize the time required to construct them by caching them.

String constraints: SMC/Kaluza

Recall the SMC-Big and SMC-Small datasets from Section 6.8.1. We ran these two datasets several more times, starting with the string length bound b at 10 characters and raising it up to 100 characters. Since our goal was to evaluate the usefulness of caching model-counting objects, we did not clear the cache between successive values of b. Again, we did not compare with Green in these experiments because Green (without Cashew) does not support constraints over strings.

Figure 6.5 shows the cumulative time spent running SMC-Big and SMC-Small, respectively, for $b \in \{10, 20, 30, \ldots, 100\}$ characters. We did this twice for each dataset. The upper lines (red) correspond to Cashew with parameterized caching disabled (model-counting objects are not cached). The lower lines (blue) correspond to Cashew with parameterized caching enabled. In this mode, an extra cost is paid to cache the model-counting objects, but doing so enables the possibility of reusing them later on. The left chart shows that caching model-counting objects

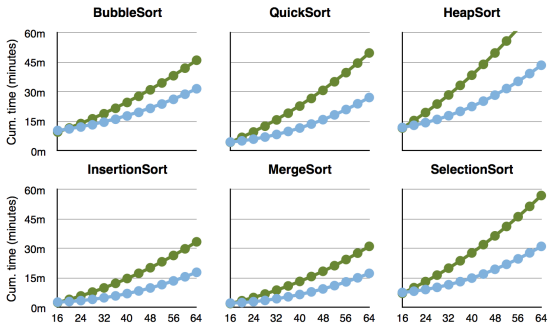


Figure 6.6: Symbolic execution of sorting/searching programs for increasing bounds. Green (above) vs. Cashew (below).

is indeed beneficial for SMC-Big. This is an idealized amortization scenario, since all stored model-counting objects are reused on each successive bound value. Nevertheless, it is useful to confirm that for this dataset, running even one additional bound is profitable, and that this profit becomes larger each time we run the dataset for an additional value of b. The right chart shows a similar phenomenon for SMC-Small, but although the gap does increase, the lines are so close together that caching model-counting objects would probably not be worth its cost. This is consistent with a large number of small problems.

Arithmetic constraints

The goal of these experiments is to evaluate the usefulness of Cashew's parameterized caching when symbolically executing Java code whose branch conditions involve linear integer arithmetic operations. Green can handle arithmetic constraints, so we can use it as the baseline for these experiments.

One well-known class of algorithms that involve integer arithmetic constraints and give rise to nontrivial path conditions are classical sorting algorithms. For these experiments we ran an SPF-based quantitative analysis (symbolic execution and model counting on complete path conditions) on the following algorithms: BubbleSort, InsertionSort, SelectionSort, QuickSort, HeapSort, and MergeSort.

Figure 6.6 shows the cumulative time spent in the analysis of each of the seven Java programs, for $b \in \{16, 20, 24, \ldots 64\}$. Since we are counting over the integers, the bound b now denotes the maximum number of bits that may be used to represent an integer. We ran each series twice. The upper curve (green) corresponds to Green, with caching enabled, using its normalization procedure for integer arithmetic constraints. The lower curve (blue) corresponds to Cashew with parameterized caching enabled.

The magnitude of the gap between both curves varies for different programs. In most cases, the initial run on an empty cache (for $b = 16$) is slightly more costly for Cashew due to the overhead of having to store all the model-counting objects in the cache. This is compensated as soon as they are reused at least once, and in all cases we see that the gap between the curves grows as the model-counting objects are reused further. This confirms that parameterized caching is beneficial for these programs if there is a reasonable chance that the model-counting objects may be reused.

6.9 Related Work

Our work builds on top of Green [127], an external framework for caching the results of calls to satisfiability solvers or model checkers developed by Visser et al. Green has been extended by Jia et. al in their tool GreenTrie [128], which, for a given target formula, efficiently queries the cache for satisfiable formulas that imply it or unsatisfiable formulas implied by it. This allows for additional resuse when GreenTrie is able to detect an implication relation between the target constraint and one in the database. Another caching framework Recal [129], transforms a linear integer arithmetic constraint to a matrix, canonizes it, and uses the result as a normal form with which the query the database. Like GreenTrie, Recal is able to detect some implications between constraints. Kopp et al. [152] also develop a framework for caching, which like ours, uses a group theoretic framework to define a normal form for constraints.

Cashew differs notably from these previous caching frameworks. First, we present a parameterized model counting approach for quantative program analysis which allows us to cache and reuse a model-counter in addition to the results of model counting queries. This allows us to reuse results for model countring queries across different bounds. Cashew also exploits more expressive normalization techniques with reductions that preserve only the *number* of solutions of a constraint instead of their solution set. This allows us to reuse information that the above caching frameworks could not. Cashew is also able to handle string constraints, extending its applicablity to analyses over string manipulating code. In constrant, Green, GreenTrie and Recal only support caching over the domain of quantifier-free linear integer arithmetic. The work of Kopp et al. is built over a domain of first order formulas restricted to predicates, variables, quantifiers and logical connectives but no constants or function symbols and their framework is not implemented.

6.10 Conclusions

In this chapter, I provided a general group-theoretic framework for constraint normalization, and presented constraint normalization techniques for string constraits, arithmetic constraints and their combinations. I extended constraint caching to string constraints and combinations of string and arithmetic constraints. I presented constraint normalization techniques for quantitative program analysis that preserve the cardinality of the solution set for a given constraint but not necessarily the solution set itself. I presented parameterized constraint caching techniques that can reuse the result of a previous model counting query even if the bounds of the queries do not match. The experiments demonstrate that, when combined with our constraint normalization approach, constraint caching can significantly improve the performance of quantitative program analyses critical to understanding the severity of side-channel vulnerabilities.

Bibliography

[1] J. Friedman, *Tempest: A signal problem*, *NSA Cryptologic Spectrum* **35** (1972) 76.

[2] D. Brumley and D. Boneh, *Remote timing attacks are practical*, *Computer Networks* **48** (2005), no. 5 701–716.

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

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

[5] `https://github.com/Apogee-Research/STAC/tree/master/Engagement_Challenges`.

[6] "Space/time analysis for cybersecurity (stac)." `https://www.darpa.mil/program/space-time-analysis-for-cybersecurity`.

[7] R. T. Prosser, *Applications of boolean matrices to the analysis of flow diagrams*, in *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pp. 133–138, ACM, 1959.

[8] E. S. Lowry and C. W. Medlock, *Object code optimization*, *Communications of the ACM* **12** (1969), no. 1 13–22.

[9] M. S. Hecht and J. D. Ullman, *Flow graph reducibility*, in *Proceedings of the fourth annual ACM symposium on Theory of computing*, pp. 238–250, ACM, 1972.

[10] `https://github.com/vlab-cs-ucsb/coco-channel`.

[11] D. Balasubramanian, Z. Zhang, D. McDermet, and G. Karsai, *Janalyzer: A static analysis tool for java bytecode*, .

[12] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, *Taj: effective taint analysis of web applications*, in *ACM Sigplan Notices*, vol. 44, pp. 87–97, ACM, 2009.

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

[14] "Z3 api in python."

[15] S. Anand, C. Păsăreanu, and W. Visser, *Jpf–se: A symbolic execution extension to java pathfinder*, *Tools and Algorithms for the Construction and Analysis of Systems* (2007) 134–138.

[16] S. Chiba, *Javassist—a reflection-based programming wizard for java*, in *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, vol. 174, 1998.

[17] V. Hindriksen, "How expensive is an operation on a cpu."

[18] M. Joye and S.-M. Yen, *The montgomery powering ladder*, in *CHES*, vol. 2, pp. 291–302, Springer, 2002.

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

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

[21] D. Genkin, I. Pipman, and E. Tromer, *Get your hands off my laptop: Physical side-channel key-extraction attacks on pcs*, *Journal of Cryptographic Engineering* **5** (2015), no. 2 95–112.

[22] P. C. Kocher, *Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems*, in *Annual International Cryptology Conference*, pp. 104–113, Springer, 1996.

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

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

[25] Q.-S. Phan, L. Bang, C. S. Pasareanu, P. Malacaria, and T. Bultan, *Synthesis of adaptive side-channel attacks.*, *IACR Cryptology ePrint Archive* **2017** (2017) 401.

[26] K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen, *Sidebuster: automated detection and quantification of side-channel leaks in web application development*, in *Proceedings of the 17th ACM conference on Computer and communications security*, pp. 595–606, ACM, 2010.

[27] J. Agat, *Transforming out timing leaks*, in *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 40–53, ACM, 2000.

[28] D. Hedin and D. Sands, *Timing aware information flow security for a javacard-like bytecode*, *Electronic Notes in Theoretical Computer Science* **141** (2005), no. 1 163–182.

[29] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, *Cacheaudit: A tool for the static analysis of cache side channels*, *ACM Transactions on Information and System Security (TISSEC)* **18** (2015), no. 1 4.

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

[31] L. Mather and E. Oswald, *Quantifying side-channel information leakage from web applications.*, *IACR Cryptology ePrint Archive* **2012** (2012) 269.

[32] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, *Diffuzz: Differential fuzzing for side-channel analysis*, *arXiv preprint arXiv:1811.07005* (2018).

[33] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, *et. al.*, *The worst-case execution-time problem—overview of methods and survey of tools*, .

[34] Y.-T. S. Li and S. Malik, *Performance analysis of embedded software using implicit path enumeration*, in *ACM SIGPLAN Notices*, vol. 30, pp. 88–98, ACM, 1995.

[35] P. P. Puschner and A. V. Schedl, *Computing maximum task execution times—a graph-based approach*, *Real-Time Systems* **13** (1997), no. 1 67–91.

[36] Apache, "Apache Shiro." `http://shiro.apache.com/`.

[37] GraphHopper, "GraphHopper." `http://www.graphhopper.com/`.

[38] J. Aycock, *A brief history of just-in-time*, *ACM Comput. Surv.* **35** (June, 2003) 97–113.

[39] D. E. Knuth, *An empirical study of fortran programs*, *Software: Practice and experience* **1** (1971), no. 2 105–133.

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

[41] J. McCarthy, *Recursive functions of symbolic expressions and their computation by machine*, *Communications of the ACM* **3** (1960), no. 4 184–195.

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

[43] G. J. Hansen, *Adaptive systems for the dynamic run-time optimization of programs*, tech. rep., 1974.

[44] P. S. Abrams, *An apl machine*, tech. rep., 1970.

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

[46] JVM Team, *The Java HotSpot Virtual Machine*, tech. rep., Technical White Paper, Sun Microsystems, 2006.

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

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

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

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

[51] C. Newland, "The JITWatch tool." `https://github.com/AdoptOpenJDK/jitwatch`.

[52] OpenStreetMap, "OpenStreetMap." `https://www.openstreetmap.org/`.

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

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

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

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

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

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

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

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

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

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

[63] Y. Xiao, M. Li, S. Chen, and Y. Zhang, *Stacco: Differentially analyzing side-channel traces for detecting ssl/tls vulnerabilities in secure enclaves*, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, (New York, NY, USA), pp. 859–874, ACM, 2017.

[64] N. Rosner, I. B. Kadron, L. Bang, and T. Bultan, *Profit: Detecting and Quantifying Side Channels in Networked Applications*, 2018.

[65] S. Guo, M. Wu, and C. Wang, *Adversarial symbolic execution for detecting concurrency-related cache timing leaks*, in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 377–388, ACM, 2018.

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

[67] C. S. Pasareanu, Q. Phan, and P. Malacaria, *Multi-run side-channel analysis using symbolic execution and max-smt*, in *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pp. 387–400, IEEE Computer Society, 2016.

[68] L. Bang, N. Rosner, and T. Bultan, *Online synthesis of adaptive side-channel attacks based on noisy observations*, in *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, pp. 307–322, April, 2018.

[69] S. Saha, I. B. Kadron, W. Eiers, L. Bang, and T. Bultan, *Attack synthesis for strings using meta-heuristics*, *SIGSOFT Softw. Eng. Notes* **43** (Jan., 2019) 56–56.

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

[71] D. Page, *Theoretical use of cache memory as a cryptanalytic side-channel.*, *IACR Cryptology ePrint Archive* **2002** (2002) 169.

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

[73] C. Percival, *Cache missing for fun and profit*, 2005.

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

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

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

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

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

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

[80] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, D. Ponomarev, *et. al.*, *Branchscope: A new side-channel attack on directional branch predictor*, in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 693–707, ACM, 2018.

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

[82] M. Zalewski, *American fuzzy lop (afl) fuzzer*, *http://lcamtuf.coredump.cx/afl/* (2017).

[83] D. Balasubramanian, Z. Zhang, D. McDermet, and G. Karsai, *Janalyzer: A static analysis tool for java bytecode*, *ISIS* **17** (2017) 104.

[84] S. Horwitz, T. Reps, and D. Binkley, *Interprocedural slicing using dependence graphs*, in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, (New York, NY, USA), pp. 35–46, ACM, 1988.

[85] J. Ferrante, K. J. Ottenstein, and J. D. Warren, *The program dependence graph and its use in optimization*, *ACM Trans. Program. Lang. Syst.* **9** (July, 1987) 319–349.

[86] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 2015.

[87] E. Bruneton, R. Lenglet, and T. Coupaye, *Asm: A code manipulation tool to implement adaptable systems*, in *In Adaptable and extensible component systems*, 2002.

[88] R. Kersten, K. Luckow, and C. S. Păsăreanu, *Poster: Afl-based fuzzing for java with kelinci*, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2511–2513, ACM, 2017.

[89] M. Zalewski, *The bug-o-rama trophy case*, *http://lcamtuf.coredump.cx/afl/#bugs* (2017).

[90] "Apache ftpserver." `https://mina.apache.org/ftpserver-project/`. Accessed: 2018-08-21.

[91] "Authentication plugin for the bukkit/spigot api." `https://github.com/AuthMe/AuthMeReloaded`. Accessed: 2018-08-21.

[92] T. A. Proebsting, G. M. Townsend, P. G. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson, *Toba: Java for applications-a way ahead of time (wat) compiler.*, in *COOTS*, pp. 41–54, 1997.

[93] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko, *Compiling java just in time*, *IEEE Micro* **17** (1997), no. 3 36–43.

[94] "Hotspot glossary of terms." `https://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html`.

[95] "OpenJDK: JDK 14 source code (Mercurial repository), tag b132.." `http://hg.openjdk.java.net/jdk/jdk14/jdk`.

[96] S. Dever, S. Goldman, and K. Russell, *New compiler optimizations in the java hotspot virtual machine*, in *JavaOne Conference*, 2006.

[97] "Jvm performance optimization, part 2: Compilers." `https://www.infoworld.com/article/2078635/jvm-performance-optimization-part-2-compilers.html`.

[98] "Jvm performance magic tricks." `https://blog.overops.com/jvm-performance-magic-tricks/`.

[99] "Java on steroids: 5 super useful jit optimization techniques." `https://dzone.com/articles/java-on-steroids-5-super-useful-jit-optimization-t/`.

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

[101] M. Backes, B. Köpf, and A. Rybalchenko, *Automatic discovery and quantification of information leaks*, in *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pp. 141–153, 2009.

[102] C. Barrett, L. de Moura, S. Ranise, A. Stump, and C. Tinelli, *The smt-lib initiative and the rise of smt*, in *Haifa Verification Conference*, pp. 3–3, Springer, 2010.

[103] C. Barrett, M. Deters, L. De Moura, A. Oliveras, and A. Stump, *6 years of smt-comp*, *Journal of Automated Reasoning* **50** (2013), no. 3 243–277.

[104] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, *Cvc4*, in *International Conference on Computer Aided Verification*, pp. 171–177, Springer, 2011.

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

[106] B. Dutertre, *Yices 2.2*, in *International Conference on Computer Aided Verification*, pp. 737–744, Springer, 2014.

[107] S. Khurshid, C. S. Pasareanu, and W. Visser, *Generalized symbolic execution for model checking and testing*, in *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, pp. 553–568, 2003.

[108] P. Godefroid, N. Klarlund, and K. Sen, *DART: directed automated random testing*, in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pp. 213–223, 2005.

[109] K. Sen, D. Marinov, and G. Agha, *CUTE: a concolic unit testing engine for C*, in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pp. 263–272, 2005.

[110] C. Cadar, D. Dunbar, and D. R. Engler, *KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs*, in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pp. 209–224, 2008.

[111] J. Geldenhuys, M. B. Dwyer, and W. Visser, *Probabilistic symbolic execution*, in *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pp. 166–176, 2012.

[112] M. Borges, A. Filieri, M. d'Amorim, and C. S. Pasareanu, *Iterative distribution-aware sampling for probabilistic symbolic execution*, in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pp. 866–877, 2015.

[113] K. S. Luckow, C. S. Pasareanu, M. B. Dwyer, A. Filieri, and W. Visser, *Exact and approximate probabilistic symbolic execution for nondeterministic programs*, in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014* (I. Crnkovic, M. Chechik, and P. Grünbacher, eds.), pp. 575–586, ACM, 2014.

[114] A. Filieri, C. S. Pasareanu, and W. Visser, *Reliability analysis in symbolic pathfinder*, in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pp. 622–631, 2013.

[115] D. Clark, S. Hunt, and P. Malacaria, *A static analysis for quantifying information flow in a simple imperative language*, *Journal of Computer Security* **15** (2007), no. 3 321–371.

[116] S. McCamant and M. D. Ernst, *Quantitative information flow as network flow capacity*, in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pp. 193–205, 2008.

[117] G. Smith, *On the foundations of quantitative information flow*, in *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS)*, pp. 288–302, 2009.

[118] J. Heusser and P. Malacaria, *Quantifying information leaks in software*, in *Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010*, pp. 261–269, 2010.

[119] Q. Phan, P. Malacaria, O. Tkachuk, and C. S. Pasareanu, *Symbolic quantitative information flow*, *ACM SIGSOFT Software Engineering Notes* **37** (2012), no. 6 1–5.

[120] Q. Phan, P. Malacaria, C. S. Pasareanu, and M. d'Amorim, *Quantifying information leaks using reliability analysis*, in *Proceedings of the International Symposium on Model Checking of Software, SPIN 2014, San Jose, CA, USA*, pp. 105–108, 2014.

[121] Q.-S. Phan and P. Malacaria, *Abstract model counting: a novel approach for quantification of information leaks*, in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pp. 283–292, ACM, 2014.

[122] B. Mao, W. Hu, A. Althoff, J. Matai, J. Oberg, D. Mu, T. Sherwood, and R. Kastner, *Quantifying timing-based information flow in cryptographic hardware*, in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 552–559, IEEE Press, 2015.

[123] C. G. Val, M. A. Enescu, S. Bayless, W. Aiello, and A. J. Hu, *Precisely measuring quantitative information flow: 10k lines of code and beyond*, in *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pp. 31–46, IEEE, 2016.

[124] V. Baldoni, N. Berline, J. D. Loera, B. Dutra, M. Köppe, S. Moreinis, G. Pinto, M. Vergne, and J. Wu, "Latte integrale v1.7.2." http://www.math.ucdavis.edu/ latte/, 2004.

[125] L. Luu, S. Shinde, P. Saxena, and B. Demsky, *A model counter for constraints over unbounded strings*, in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, p. 57, 2014.

[126] A. Aydin, L. Bang, and T. Bultan, *Automata-based model counting for string constraints*, in *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)*, pp. 255–272, 2015.

[127] W. Visser, J. Geldenhuys, and M. B. Dwyer, *Green: reducing, reusing and recycling constraints in program analysis*, in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, p. 58, ACM, 2012.

[128] X. Jia, C. Ghezzi, and S. Ying, *Enhancing reuse of constraint solutions to improve symbolic execution*, in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 177–187, ACM, 2015.

[129] A. Aquino, F. A. Bianchi, M. Chen, G. Denaro, and M. Pezzè, *Reusing constraint proofs in program analysis*, in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 305–315, ACM, 2015.

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

[131] P. Hooimeijer and W. Weimer, *A decision procedure for subset constraints over regular languages*, in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 188–198, 2009.

[132] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, *Hampi: a solver for string constraints*, in *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)*, pp. 105–116, 2009.

[133] P. Hooimeijer and W. Weimer, *Solving string constraints lazily*, in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 377–386, 2010.

[134] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, *A symbolic execution framework for javascript*, in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[135] V. Ganesh, M. Minnes, A. Solar-Lezama, and M. C. Rinard, *Word equations with length constraints: What's decidable?*, in *Proceedings of the 8th International Haifa Verification Conference (HVC)*, pp. 209–226, 2012.

[136] Y. Zheng, X. Zhang, and V. Ganesh, *Z3-str: A z3-based string solver for web application analysis*, in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pp. 114–124, 2013.

[137] G. Li and I. Ghosh, *PASS: string solving with parameterized array and interval automaton*, in *Proceedings of the 9th International Haifa Verification Conference (HVC)*, pp. 15–31, 2013.

[138] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman, *String constraints for verification*, in *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, pp. 150–166, 2014.

[139] T. Liang, N. Tsiskaridze, A. Reynolds, C. Tinelli, and C. Barrett, *A decision procedure for regular membership and length constraints over unbounded strings*, in *Proceedings of the 10th International Symposium on Frontiers of Combining Systems* (C. Lutz and S. Ranise, eds.), vol. 9322 of *Lecture Notes in Computer Science*, pp. 135–150, Springer, 2015.

[140] T. Liang, A. Reynolds, N. Tsiskaridze, C. Tinelli, C. Barrett, and M. Deters, *An efficient smt solver for string constraints*, *Formal Methods in System Design* **48** (2016), no. 3 206–234.

[141] M. Trinh, D. Chu, and J. Jaffar, *S3: A symbolic string solver for vulnerability detection in web applications*, in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 1232–1243, 2014.

[142] J. A. D. Loera, R. Hemmecke, J. Tauzer, and R. Yoshida, *Effective lattice point counting in rational convex polytopes*, *Journal of Symbolic Computation* **38** (2004), no. 4 1273 – 1302.

[143] S. Chakraborty, K. S. Meel, R. Mistry, and M. Y. Vardi, *Approximate probabilistic inference via word-level counting*, *arXiv preprint arXiv:1511.07663* (2015).

[144] M. Thurley, *sharpsat–counting models with advanced component caching and implicit bcp*, in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 424–429, Springer, 2006.

[145] S. Verdoolaege, *barvinok: User guide*, *Version 0.23), Electronically available at http://www. kotnet. org/˜ skimo/barvinok* (2007).

[146] J. Crawford, *A theoretical analysis of reasoning by symmetry in first-order logic*, in *AAAI Workshop on Tractable Reasoning*, Citeseer, 1992.

[147] I. P. Gent and B. Smith, *Symmetry breaking during search in constraint programming.* Citeseer, 1999.

[148] J. Crawford, M. Ginsberg, E. Luks, and A. Roy, *Symmetry-breaking predicates for search problems*, *KR* **96** (1996) 148–159.

[149] J. Rizzo and T. Duong, *The crime attack*, Ekoparty Security Conference, 2012.

[150] M. Weir, S. Aggarwal, M. P. Collins, and H. Stern, *Testing metrics for password creation policies by attacking large sets of revealed passwords*, in *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pp. 162–175, 2010.

[151] "Redis." `https://redis.io/`.

[152] T. Kopp, P. Singla, and H. Kautz, *Toward caching symmetrical subtheories for weighted model counting*, in *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.