The Dissertation Committee for Zhiqiang Zang
certifies that this is the approved version of the following dissertation:

**Template-Based Testing for Java Just-in-Time Compilers**

**Committee**:

Milos Gligoric, Supervisor

Christine Julien

Owolabi Legunsen

Christopher Rossbach

August Shi

# Template-Based Testing for Java Just-in-Time Compilers

**by**

**Zhiqiang Zang**

**Dissertation**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Doctor of Philosophy**

**The University of Texas at Austin**
**May 2024**

# Acknowledgments

It has been a long six years. I will forever cherish this remarkable journey—for the blood, toil, tears, and sweat invested in every day and night. However, there is no way I could have made it on my own without the amazing support from so many awesome people I have encountered along the way. I apologize in advance if your name slips my mind here due to my own bad memory. Nonetheless, you have my gratitude.

I would like to thank my advisor, Milos Gligoric, who has been an exceptional guide throughout my Ph.D. journey. Milos taught me comprehensive knowledge about research, coding, and writing, and he has been constantly encouraging me to strive for my best. I vividly recall the extensive time we spent refining slides and tirelessly practicing before my first conference talk. While it left me nearly exhausted, I now appreciate his resolute commitment, as it led to a successful presentation and earned praises from the audience. I am deeply thankful for Milos' outstanding support and guidance over the years that helped me grow into a professional researcher and programmer. I am confident that future students will also come to appreciate his passion for hacking, sense of humor, and the occasional promise (fulfilled, unless the snacks were forgotten in the trunk) to bring treats to the group.

I would also like to thank my committee members: Christine Julien, Owolabi Legunsen, Christopher Rossbach, August Shi. Their invaluable feedback significantly enriched the presentation of my work in this dissertation.

In particular, I am grateful to August, who not only provided guidance on the JAttack project but also played a crucial role in shaping my first first-author paper [144]. I also want to thank August, Owolabi, and Milos for hosting a weekly seminar, which provided me with opportunities to engage with fellow Ph.D. students with whom I enjoyed several conferences together.

I feel fortunate that I had an opportunity to collaborate with a remarkable group of individuals, including Ben Buhse, Ethan Beyer, Sarfraz Khurshid, Aleksandar Milicevic, Pengyu Nie, Marinela Parovic, August Shi, Aditya Thimmaiah, Nathaniel Wiatrek, Thomas Wei, Fu-Yao Yu. Their support and contributions were indispensable to the work I did.

A special thanks to Pengyu, who recommended me to Milos before I joined as a Ph.D. student in 2018. Pengyu also taught me valuable technical and research skills during my initial two years in graduate school, laying a solid foundation for the rest of my Ph.D. journey.

I am also grateful for the outstanding collaboration with Fu-Yao, an exceptional undergraduate student. His productivity and self-motivation surpass even my own, and over the last two years, we have worked closely together, resulting in the publication of several papers.

I owe my thanks to my lovely officemates: Nader Al Awar, Abdelrahman Baz, Ahmet Celik, Changyong Hu, Yang Hu, Jaeseong Lee, Chengpeng Li, Yu Liu, Pengyu Nie, Marinela Parovic, Shanto Rahman, Aditya Thimmaiah, Wenxi Wang, Jiyang Zhang, Xiong Zheng, Chenguang Zhu. Chatting and having a good time with them is my primary motivation for heading to the office each day. (Well, the promise of Milos' snacks might be a close second.)

I want to give a special shoutout to Jiyang for being an excellent companion. Jiyang never grew tired of responding to my seemingly meaningless jokes (or random words) from the seat behind me. In the unpredictable times during the pandemic, Jiyang's constant presence was the one comfort in my daily life.

I also have many friends outside the office: Jin Han, Zhixuan Huan, Dawei Liang, Haotian Wang, Yating Wu, Jiayi Yang, Bo Zhang, Yuhao Zheng, etc. Together, we have created countless wonderful memories, and I appreciate every second spent with them—whether in restaurants, in cars on the road, by streets and lakes, or under trees, camps, and night skies. I would like to thank my friends in China: Weihao

Li, Donglin Miao, Cong Pan, Huibin Zhou. Despite the physical distance, we have stayed connected through phone calls, messages, and online gaming. I am hopeful that I will soon have the chance to return to China and hang out with them in person! I also want to acknowledge the friends I have connected with on the Internet, including fans in the Red Alert community and fellow podcast enthusiasts. All these friendships blended more colors and sweetness in my Ph.D. journey. They expanded my horizons, inspired me to pick up new hobbies, and made me a better person.

I would like to give a big thank you to my mother, Xia Qin, and father, Zhengqing Jia. Their unwavering respect and support have allowed me to pursue my dreams without any reservations or worries in the background.

I had four outstanding summer internships. My thanks go to all my mentors, managers and colleagues who contributed to my professional growth: Zhuo Cheng and Zhen Wang at NIO, Mukul R. Prasda and Hiroaki Yoshida at Fujitsu, Xiangyang Guo, Chen Li, and Yating Zhou at Meta, and Yuxin Wang, Sonal Mahajan, Lazaro Clapp, and Raj Barik at Uber. Their guidance and insights have broaden my understanding of various aspects of the industry, and equipped me with the knowledge and skills essential for my future career.

The support staff at the Department of Electrical and Computer Engineering are the unsung heroes who make my academic life much smoother. I would like to thank Thomas Atchity, Cayetana Garcia, Melanie Gulick, Barry Levitch, and Melody Singleton for their invaluable assistance, from timely reminders about semester registrations to efficiently handling travel reimbursements.

Parts of this dissertation were published at the International Conference on Automated Software Engineering (ASE) 2022 [144] and the International Conference on Software Engineering (ICSE) 2023 [146] (Chapter 2), as well as the International Conference on the Foundations of Software Engineering (FSE) 2024 (Chapter 3). I would like to thank the anonymous reviewers and the audiences at these conferences for their invaluable comments. It is an honor that my ASE 2022 paper on JAT-

tack [144] received an Association for Computing Machinery Special Interest Group on Software Engineering (ACM SIGSOFT) Distinguished Paper Award.

I am thankful for the financial support that made my research possible, including a Google Faculty Research Award, a grant from the Army Research Office, and funding from the US National Science Foundation under Grant Nos. CCF-1652517, CCF-2107291, CCF-2145774, and CCF-2217696.

# Abstract

## Template-Based Testing for Java Just-in-Time Compilers

Zhiqiang Zang, PhD
The University of Texas at Austin, 2024

SUPERVISOR: Milos Gligoric

Compilers are among the most critical components in the software development toolchain, and their correctness is of utmost importance. A bug in a compiler might lead to a crash during the translation, an incorrect output (native code does not match the semantics of the program written by developers), or even expose security vulnerabilities in the generated code.

Compiler developers have written thousands of tests, including programs in the compiler's target programming language, as to check for correctness. Although manual tests nicely capture developers' intuition of what programs are expected to trigger corner cases in a compiler, it is time-consuming to write a large number of such tests.

Numerous automated compiler testing techniques exist, aiming to generate extensive tests. They broadly fall into two categories: grammar-based methods, which build tests from grammar rules, and mutation-based techniques, which build tests by mutating seed programs. However, they offer limited room for compiler developers to embed their domain knowledge into the testing process. Particularly, these tools are ineffective in discovering bugs in just-in-time (JIT) compilers. JIT compilers, dynamically (i.e., at runtime) rewrite parts of programs to optimize program execution based on profiling data. Testing such compilers requires carefully crafted inputs

that trigger JIT compilation and provide challenging code snippets for optimizing compilers.

This dissertation presents two frameworks to combine developers' domain knowledge (via templates) with automated testing to detect bugs in JIT compilers.

The dissertation first introduces JAttack, a framework that enables template-based testing for compilers. Using JAttack, a developer writes a template program that describes a set of programs to be generated and given as test inputs to a compiler. Such a framework enables developers to incorporate their domain knowledge on testing compilers, giving a basic program structure that allows for exploring complex programs that can trigger sophisticated compiler optimizations. A developer writes a template program in the host language (Java) that contains holes to be filled by JAttack. JAttack generates programs by executing templates and filling each hole by randomly choosing expressions and values, available within the search space defined by the hole. We demonstrate JAttack's capabilities in helping test Java JIT compilers. Using JAttack, we have found seven bugs in HotSpot that were confirmed and fixed by Oracle developers. Five of them were previously unknown, including two unknown CVEs (Common Vulnerabilities and Exposures).

This dissertation then introduces LeJit, an overarching framework wrapping JAttack for testing Java JIT compilers. LeJit automatically creates template programs from existing Java code by converting expressions to holes. To make created templates executable, LeJit also generates necessary glue code that creates instances used as arguments to methods in templates. To obtain instances of complex types (i.e., non-primitive types) needed for created templates, LeJit captures instances of various types during testing of methods from which templates are to be extracted. Using LeJit, we have found 15 bugs in popular Java JIT compilers, including five bugs in HotSpot, nine bugs in OpenJ9, and one bug in GraalVM. All of these bugs have been confirmed by Oracle and IBM developers, 11 of which were previously unknown, including two unknown CVEs.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1: Introduction

Compilers play a pivotal role in the software development process, as indispensable components of the software development toolchain. They hold a position of significant trust within the software development community, emphasizing the criticality of their correctness. A single bug within a compiler can precipitate a cascade of issues, potentially resulting in program crashes [42], the generation of erroneous output that deviates from the intended program semantics [95], or even the exposure of security vulnerabilities within the resultant code [128].

However, the correct implementation of compilers is never easy because of their complexity. A typical compiler consists of a number of interacting components, e.g., lexer, parser, optimizer, code generation, etc [2]. The correct implementation requires precise understanding of all these components. Moreover, the input and output of compilers are programs that can have complex structures. Thus, reasoning about the behavior of a compiler is all but trivial.

The difficulty of implementing a compiler brings unique challenges to testing the implementation. First, the semantic richness of the input and output makes compilers usually lack a formal specification. While the high-level goal, which is to translate a program in the source language into a semantics-preserving program in the target language, the low-level details are usually unspecified. For example, when to apply what optimizations is rarely specified, making it difficult to check whether a compiler applies all desired optimizations. Second, the extremely large input and output domains allow a small change in the input program to make a huge difference in the expected behavior of the compiler. Thus, meticulously designed inputs are necessary to test desired parts of compilers. For example, changing a single value in the loop condition may cause the compiler to stop performing loop optimizations. Third, it is not trivial for a program to reach deeper components of compilers, such as optimizer or code generation, because the program has to pass all the checks in the lexer and parser. Therefore, non-trivial work, e.g., following language grammars, is required to create these programs if deep testing of compilers is desired.

Compiler developers have written thousands of *tests*, including programs in the compiler's target programming language, as to check for correctness [50, 86, 96]. While these hand-written tests nicely capture *developers' intuition* into scenarios that may expose corner cases, the process of generating a substantial quantity of such tests is notably time-consuming. A prior study showed that close to 50% of developers' time is spent on testing [37].

As a result, researchers and practitioners have developed several automated techniques for testing compilers, namely generating a large number of tests to feed to a compiler as inputs. Traditionally, the compiler testing landscape has predominantly revolved around static compilers, e.g., well known GCC, LLVM, and javac. Testing tools such as Csmith [139] and Hephaestus [22] have effectively discovered bugs within these static compilers [140, 141]. However, these testing tools, effective as they may be in their domain, fall short when it comes to the intricate task of identifying bugs within just-in-time (JIT) compilers [8].

JIT compilers, JIT for short, operate dynamically, rewriting parts of programs at runtime to optimize their execution based on profiling data [8]. This unique functionality demands a specialized approach to testing, one that involves crafting meticulously designed inputs capable of triggering JIT compilation and presenting complex code structures that challenge optimizing compilers.

Recent compiler testing techniques, including JIT compiler testing, mainly fall in two categories: grammar-based [63, 84, 139, 143] and mutation-based [32, 33, 131, 153]. In the former category, programs are generated from scratch following the production rules specified in the language grammar. In the latter category, the process usually starts with initial seed programs that are subsequently genetically mutated. While existing approaches are valuable, they do have shortcomings. They provide limited ways for compiler developers to fully integrate their expertise and

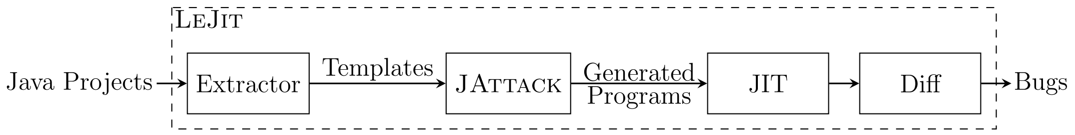

Figure 1.1: The overview of JAttack and LeJit.

domain knowledge into the testing process.

*We believe that integrating developers' domain knowledge into automated testing can significantly enhance current JIT compiler testing techniques.* This dissertation presents two primary research endeavors with this objective: (1) a template-based compiler testing framework that empowers developers to write templates and generates from these templates numerous concrete programs as test inputs to compilers, and (2) a unified automated testing framework that streamlines creation of templates.

Figure 1.1 shows the overview of JAttack and LeJit. First, we introduce JAttack, a framework that enables *compiler testing using templates.* Using JAttack, a developer writes a *template program* (template for short) that outlines a *collection of concrete programs* to serve as test inputs to a compiler. Unlike prior work, our framework enables developers to express richer manual tests for compilers. Our design of a template effectively captures the developers' intuition, in very much the same way as hand-written tests, but it goes a step further by offering flexibility to express variants of those tests that can be obtained by *executing* the templates. (Figure 1.2 shows an example template, which is discussed in detail in Section 2.2.) JAttack complements existing automated compiler-testing techniques by allowing developers to use their expertise to provide a structure of a program on which JAttack can further explore.

In JAttack, a developer writes a template in the host language (Java), which contains *holes*, i.e., placeholders, to be filled by JAttack. Each hole is written in a domain-specific language (DSL) fully embedded in the host language. Namely, we do not change the syntax, compiler, or runtime environment of the host language. The DSL we defined is a collection of APIs that enables developers to specify characteris-

```java
import static jattack.Boom.*;
public class C {
  static int s1;
  static int s2;
  @Entry
  public static int m() {
    int[] arr1 = { s1++, s2, intVal().eval(), intVal().eval(),
                   intVal().eval()};
    for (int i = 0; i < arr1.length; ++i)
      if (logic(relation(intId(), intId(), LE),
              relation(intId(), intId(), LE),
              AND, OR).eval())
        arr1[i] &= arithmetic(intId(), intId(), ADD, MUL).eval();
    return 0; } }
```

Figure 1.2: An example of a template. One generated program from this template revealed a bug in the HotSpot JIT compiler.

tics of each hole they want explored in a template. Each API call defines the search space for the hole, i.e., a set of possible expressions and values. The JAttack's API ensures that each hole can be type-checked by the host compiler.

JAttack repeatedly *executes* the template (up to $N$ times) to fill each hole and as a result generates a concrete program. During the execution, when JAttack encounters a hole the first time, JAttack *randomly* chooses expressions and values available within the search space defined by the hole. Once the hole is filled, the filled expression or value is used until the end of the current execution. Next, to detect any bugs related to JIT compilers, each generated program is executed $N$ times using different Java JIT compilers, potentially detecting bugs via differential testing [87].

The benefit of JAttack, or template-based compiler testing, is that developers have full control of the space that should be tested and the way programs should evolve. However, JAttack requires substantial developers' engagement, as both template program design and hole search space definition are done manually.

To automate Java JIT compiler testing by using JAttack and providing templates automatically, we introduce another framework that is built around JAttack, dubbed LeJit, for automatically creating template programs from existing code.

LeJit creates templates by rewriting expressions to holes, as well as generating necessary glue code (e.g., code that creates instances of non-primitive types on which methods can be invoked) to make those templates executable. Execution of created templates, which is done by JAttack, randomly fills the holes to create concrete programs that are used as inputs for Java JIT compiler testing.

LeJit is designed to create a template from any existing method. One of the key challenges was to enable templates for methods that accept instances of complex types as arguments (including an instance on which the method is to be invoked). Our key insight in this direction is to capture instances of various types during testing of methods from which templates are to be extracted. These tests can be either existing manual tests or automatically generated unit tests.

Therefore, when compared to other automated testing techniques for JIT compilers, LeJit sits in between mutation-based techniques and template-based techniques. LeJit automatically creates templates from any existing code and JAttack uses the created templates to generate concrete programs to test Java JIT compilers.

The key contributions of this dissertation include:

⋆ We introduce JAttack, the framework for templating tests for compilers. JAttack is designed to complement manual tests and blend developer's intuition (via templates) and automated testing to increase likelihood to detect bugs in Java JIT compilers. We introduce a programming and an execution model to integrate templates entirely in the host language (Java), without changing the syntax or the runtime environment. Templates are like hand-written programs with holes; each hole, expressed using a DSL, specifies values that the hole can take. The holes are filled with random values through dynamic execution of templates. We implemented JAttack for the Java programming language and applied it to testing Java JIT compilers.

⋆ We designed and implemented LeJit, a framework for creating templates from existing code by converting expressions into holes and capturing instances of complex types during test execution. Captured instances enable execution using JAttack of templates that produce concrete programs used as inputs for compiler testing. We have implemented LeJit for Java and connected it with JAttack. We have also developed several variants of LeJit to help us understand the benefits of templates and captured instances used as values for arguments.

⋆ We have performed extensive evaluation of JAttack and LeJit. We have written 23 template programs using the DSL provided by JAttack. We used JAttack to test the HotSpot JIT compiler and discovered seven bugs, including two unknown CVEs (Common Vulnerabilities and Exposures). We also used LeJit to automatically extract 143,195 templates from ten open-source Java projects on GitHub. We then used LeJit to generate 886,178 concrete programs. We have discovered 15 additional bugs in popular Java JIT compilers, including five bugs in HotSpot, nine bugs in OpenJ9, and one bug in GraalVM, 11 of which are previously unknown, including two unknown CVEs. Table 1.1 shows all the detected bugs using JAttack and LeJit. Additionally, we compared JAttack and LeJit with Java* Fuzzer [9], JITfuzz [136], and JavaTailor [153], the state-of-the-art tools for testing Java runtime environments. Our results show that LeJit is complementary to the state-of-the-art techniques, which did not discover any of the bugs reported by JAttack or LeJit.

⋆ Both JAttack and LeJit are open source, and they are publicly available at `https://github.com/EngineeringSoftware/jattack` and `https://github.com/EngineeringSoftware/lejit`, respectively.

This dissertation brings a novel template-based approach for testing JIT compilers and a unified framework for automating the entire process. This shows the power of combing developers' insights with automated testing, and opens up opportunities for future compiler testing using templates.

Table 1.1: Detected bugs in HotSpot, OpenJ9 and GraalVM using JAttack and LeJit.

| JVM | Bug ID | Type | JDK Versions | CVE | Duplicates | Source of Templates |
|---|---|---|---|---|---|---|
| GraalVM | GR-45498 | Diff | 17, 20 | - | - | Extracted |
| HotSpot | JDK-8239244 | Diff | 8, 11, 13 | CVE-2020-14792 | - | Hand-written |
| | JDK-8258981 | Crash | 9, 10, 11, 15, 16 | - | JDK-8250609 | Hand-written |
| | JDK-8271130 | Crash | 8, 11, 16, 17 | CVE-2022-21305 | - | Extracted |
| | JDK-8271276 | Crash | 16, 17, 18 | - | - | Extracted |
| | JDK-8271459 | Diff | 8, 11, 16, 17, 18 | - | - | Extracted |
| | JDK-8271926 | Crash | 11, 16 | - | JDK-8268362 | Extracted |
| | JDK-8297730 | Diff | 9, 11, 17, 18, 19, 20, 21 | - | - | Extracted |
| | JDK-8301663 | Diff | 18, 19, 19.0.2 | - | JDK-8288064 | Extracted |
| | JDK-8303946 | Diff | 8, 11, 17, 19, 20, 21 | - | - | Extracted |
| | JDK-8304336 | Diff | 17, 19, 20, 21 | CVE-2023-22044 | - | Extracted |
| | JDK-8305946 | Crash | 17, 19, 20, 21 | CVE-2023-22045 | - | Extracted |
| | JDK-8325216 | Crash | 17, 18, 19, 20, 21 | - | JDK-8319793 | Extracted |
| OpenJ9 | 17066 | Crash | 8, 11, 17, 18 | - | - | Extracted |
| | 17129 | Diff | 8, 11, 17, 18 | - | - | Extracted |
| | 17139 | Diff | 8, 11, 17, 18 | - | - | Extracted |
| | 17171 | Crash | 11, 17, 18 | - | - | Extracted |
| | 17212 | Crash | 8, 11, 17, 18 | - | 15363 | Extracted |
| | 17249 | Diff | 8, 11, 17, 18 | - | - | Extracted |
| | 17250 | Diff | 17, 18 | - | - | Extracted |
| | 18802 | Crash | 8, 11, 17, 21 | - | 17045 | Extracted |
| | 18803 | Crash | 11, 17, 21 | - | - | Extracted |

# Chapter 2: Compiler Testing using Template Java Programs

In this chapter, we present JAttack, a framework that enables template-based testing for compilers. Using JAttack, a developer writes a template program that describes a set of programs to be generated and given as test inputs to a compiler. Such a framework enables developers to incorporate their domain knowledge on testing compilers, giving a basic program structure that allows for exploring complex programs that can trigger sophisticated compiler optimizations. A developer writes a template program in the host language (Java) that contains holes to be filled by JAttack. Each hole, written using a domain-specific language, constructs an extended abstract syntax tree (eAST). An eAST defines the search space for the hole, i.e., a set of expressions and values. JAttack generates programs by executing templates and filling each hole by randomly choosing expressions and values (available within the search space defined by the hole). Additionally, we introduce several optimizations to reduce JAttack's generation cost. While JAttack could be used to test various compiler features, we demonstrate its capabilities in helping test Java JIT compilers, whose optimizations occur at runtime after a sufficient number of iterations over the same code. Using JAttack, we have found seven critical bugs that were confirmed by Oracle developers. Five of them were previously unknown, including two unknown CVEs. JAttack shows the power of combining developers' domain knowledge (via templates) with random testing to detect bugs in JIT compilers. [1]

[1]Parts of this chapter are published at ASE 2022 [144] and ICSE DEMO 2023 [146]. I led the design, implementation, and evaluation of the system, as well as analyzing the data and writing papers.

## 2.1 Introduction

JAttack is a framework that enables *compiler testing using templates*. Using JAttack, a developer writes a template program that describes a *set of concrete programs* to be used as inputs to a compiler. Unlike prior work [32, 33, 63, 84, 131, 139, 143, 153], our framework enables developers to express richer manual tests for compilers. Our design of a template captures the developers' intuition in very much the same way as manual tests but provides an opportunity to express variants of those tests that can be obtained by *testing* the templates. The goal is similar to parameterized unit testing [129], where developers write unit tests that encapsulate some features they want to test in their code but have parameters that a backend framework explores as to obtain deeper testing around the insights the developers initially provide. Unlike with mutation-based fuzzers, compiler developers can use templates to specify exactly how to generate program variants. Figure 2.1 shows an example template, which is discussed in detail in Section 2.2. JAttack complements existing automated compiler-testing techniques that can provide a structure of a program on which JAttack can further build templates.

In JAttack, a developer writes a template in the host language (Java), which contains *holes* to be filled by JAttack. Each hole is written in a domain-specific language embedded in the host language, i.e., we do not change the syntax, compiler, nor runtime environment of the host language. We define the DSL as a set of APIs that allow developers to specify characteristics of the hole they want explored in a template, where each API call produces an instance of an *extended abstract syntax tree* (eAST); an eAST node bounds the search space for the hole, i.e., defines a set of possible expressions and values. As an example, consider the following API call that defines a hole: `relation(intVal(), intVal(), GT, LT).eval()`, which represents a logical relation between two integer literals (each can take any value between `Integer.MIN_VALUE` and `Integer.MAX_VALUE`) using either > (`GT`) or < (`LT`) relational operators; this hole evaluates to a boolean. Using the JAttack's API, each hole can then be type-checked by the host compiler.

JAttack is useful for augmenting testing for many complex compiler features as it leverages the developer insights from the provided templates. For our evaluation, we focus specifically on testing JIT compilers. Unlike traditional ahead-of-time compilers that translate a program into native code prior to deployment [2], JIT compilers translate the program during execution [8]. Certain optimizations only occur after executing specific program structures a sufficient number of times.

JAttack takes two inputs: (1) a template, and (2) an iteration count ($N$), i.e., the number of times each generated program will be executed in a loop to ensure that JIT compilation is triggered. JAttack generates a program by repeatedly *executing* the template (up to $N$ times) and filling each hole, when the hole is reached the first time, by *randomly* choosing expressions and values available within the search space defined by the hole. (In theory, a template can be exhaustively explored, but it is generally not feasible.) Next, to detect any JIT compiler bugs, each generated program is executed $N$ times using different JIT compilers, potentially detecting bugs via differential testing [87].

We also introduce three optimizations into JAttack to reduce the generation cost. The first optimization, *early stopping*, involves stopping after detecting that further generation would not fill any more holes. The second optimization, *hot filling*, dynamically transforms the template when a hole is reached the very first time; the API call is transformed into the concrete expression that the call would produce. The final optimization, *eager pruning*, uses a modern constraint solver (Z3 [39]) to detect holes for conditional statements (e.g., `if`) that always evaluate to a constant value.

To demonstrate JAttack's capabilities in testing JIT compilers, we wrote 23 templates. We focused on interesting Java language features and took inspiration from existing tests for the Java compiler. We report the cost of generation and execution, as well as benefits of our optimizations; our optimizations reduce the *generation time* by 99.50%. We used the generated programs as inputs to multiple commercial JIT compilers, including the HotSpot JIT compiler from Oracle JDK. Using just our own templates, we were able to discover two bugs in the HotSpot JIT compiler. We also extracted 5,419 templates from 77 open-source Java projects by translating random code into holes. We discovered five more bugs in the HotSpot JIT compiler using these extracted templates. All the seven bugs (see Table 1.1) have been confirmed and fixed by Oracle developers, including two previously unknown CVEs.

The key contributions from the JAttack work include:

- **Framework**. We introduce JAttack, the framework for templating tests for compilers. JAttack is designed to complement manual tests and blend developer's intuition (via templates) and random testing to increase likelihood to detect bugs in Java JIT compilers.
- **Programming and execution models**. We introduce a programming and an execution model to integrate templates entirely in the host language (Java), without changing the syntax or the runtime environment. Templates are like hand-written programs with holes; each hole, expressed using a DSL, builds an eAST that specifies values that the hole can take (i.e., defines a search space). We introduce three optimizations that are applied when generating programs from templates.
- **Use case**. We implemented JAttack for the Java programming language and applied it to testing Java JIT compilers. We evaluated JAttack by writing 23 template programs, and extracting 5,419 templates from 77 open-source Java projects. Our results show that the optimizations substantially reduce test generation time, making JAttack practical. Furthermore, we discovered seven bugs in the HotSpot JIT compiler, including two previously unknown CVEs.

JAttack is available at `https://github.com/EngineeringSoftware/jattack`.

```
import static jattack.Boom.*;
public class C {
  static int s1;
  static int s2;
  @Entry
  public static int m() {
    int[] arr1 = { s1++, s2, intVal().eval()❶, intVal().eval()❷,
                   intVal().eval()❸ };
    for (int i = 0; i < arr1.length; ++i)
      if (logic(relation(intId(), intId(), LE), relation(intId(), intId(), LE),
                AND, OR).eval()❹)
        arr1[i] &= arithmetic(intId(), intId(), ADD, MUL).eval()❺;
    return 0; } }
```

(a) An example of a template.

```
import static jattack.Boom.*;
public class C {
  static int s1;
  static int s2;
  @Entry
  public static int m() {
    int[] arr1 = { s1++, s2, 45350238❶, 681339300❷, 125652422❸ };
    for (int i = 0; i < arr1.length; ++i)
      if (arr1[3] <= s2 || s2 <= arr1[2]❹)
        arr1[i] &= arr1[1] * s1❺;
    return 0; } }
```

(b) An example of a generated program.

Figure 2.1: An example of a template and one generated program from the template.

## 2.2 Example

Figure 2.1a shows a template program that we wrote while developing JATTACK for Java. Our motivation for this template was to exercise Java JIT compiler optimizations for programs that use local arrays and static variables. It is important to note that *every template for* JATTACK *is a valid Java program.* This template uses static methods (e.g., `logic`) that are defined in the `jattack.Boom` class. As such, the Java compiler can also type-check the template.

The template contains five *holes* representing places where JATTACK should generate expressions, filling them in to create a concrete *generated program*. Three holes are between lines 7 and 8, one between lines 10 and 11, and one on line 12 (in Figure 2.1a). The number of holes is equal to the number of `eval` invocations. The `eval` invocation as well as the type information of the expression calling the `eval` allows JAttack to tell a hole from actual code.

The first three holes are defined by the `intVal` method calls; each call to `intVal` represents a hole that will be filled by an integer literal. Note that without any arguments, `intVal` produces an integer between `Integer.MIN_VALUE` and `Integer.MAX_VALUE`. The next hole (lines 10-11) defines a logical "and" or "or" expression (`logic` with the `AND` and `OR` arguments) between two relational expressions. Each relational expression (`relation`) connects two free integer variables `intId`, which can be `s1`, `s2`, `i`, or any element of `arr1` (the array index is randomly picked between 0 and the size of the array) at this point, using the `<=` operator (`LE`). The final hole (line 12) is an arithmetic expression (`arithmetic`) of two free integer variables, which are combined using either a + or a $*$ operator (`ADD` or `MUL`). We describe more details on what type of expressions we support for holes along with our API in Section 2.3.

JAttack generates programs through an *execution-based* model. In other words, JAttack fills the holes in a template after executing the template. (Unlike static generation, an execution-based model prunes the search space by only filling holes reached during execution. See Section 2.6.1 for details.) A template must have a template entry method, annotated with `@Entry` as shown in the example (method `m`). When the execution reaches any unfilled hole, JAttack generates a valid expression for that hole based on the used API calls. When all reachable holes are filled (see Section 2.3.3 for how this is determined), JAttack outputs the corresponding generated program. JAttack then calls the template entry method again to generate the next program up to the specified maximum number of programs. An example of a generated program that JAttack outputs for the template in Figure 2.1a is shown in Figure 2.1b. The numbered circles in the generated program correspond to the same ones next to holes in Figure 2.1a.

Finally, to detect any bugs related to JIT compilers, JAttack *executes* each generated program starting from the same entry method a large number of times using different JIT compilers, potentially detecting bugs via differential testing [87]. The large number of re-executions is necessary as to trigger JIT compiler optimizations. In Java, a JVM starts executing a program with an interpreter and monitors the execution for "hot" code sections, i.e., code that is frequently executed. The JIT compiler then optimizes "hot" sections. Through repeated executions of the generated method `m`, the generated program shown in Figure 2.1b revealed a bug in the HotSpot JIT compiler, crashing the JVM.

## 2.3 JAttack Framework

JAttack introduces test templating, a way to define a *set of programs* used for testing compilers. We designed JAttack guided by the following requirements: (1) developers decide where the holes should be placed and bound the search space of each hole, and (2) the domain-specific language for writing holes is non-intrusive, i.e., it requires no changes to the host compiler.

In this section, we describe our programming and execution models (Section 2.3.1), implementation for Java (Section 2.3.2), the generation procedure (Section 2.3.3), the optimizations for generation (2.3.4), and the overall JIT compiler testing procedure (Section 2.3.5).

### 2.3.1 Programming and Execution Models

We define the syntax and operational semantics of a simple imperative language with an extension to support templates. Note the language shown here includes only integer type for ease of presentation; we greatly extend the scope in our implementation for Java. The simple imperative language and extensions represent the foundations for supporting templates for general imperative languages, and our implementation in Java, described in later sections, is based on these extensions.

```
Stmts := (LABEL:)? Stmt ";" Stmts | ϵ
Stmt := AssignStmt | IfStmt | GotoStmt | halt
AssignStmt := ID "=" Exp
IfStmt := "if" "(" Exp ")" LABEL
GotoStmt := goto LABEL

Exp := ExpBasic | "⟦" ExpAlt "⟧"
ExpBasic := ExpBasic Op Operand | Operand
ExpAlt := ExpAlt "," ExpBasic | ExpBasic

Operand := ID | NUM | "⟦c⟧" | "⟦v⟧"
Op := "+" | "-" | "||" | "&&" | "==" | "!=" | "<"
```

Figure 2.2: Syntax for an imperative language with holes.

**Syntax**. Figure 2.2 defines the syntax of the simple language. A program is a sequence of zero or more statements. Each statement is either an assignment, conditional, goto, or halt. An expression in a program can combine relational, arithmetic, and logical operators. On top of these basic imperative features, the language also introduces the concept of a hole, denoted with $[\![]\!]$. These holes can be used around a sequence of comma-separated (`ExpAlt` in the figure) expressions, or they can be around individual operands, where ⟦c⟧ represents a hole for a literal/constant and ⟦v⟧ represents a hole for a variable.

**Semantics (core language)**. Valid programs can only use integer literals. We define the state of a program with the following configuration: $\langle pc, I, M, L\rangle$, where $pc$ is the program counter (initially 0), $I$ is the instruction memory (i.e., mapping from the program counter to statements or expressions), $M$ is the main memory (i.e., mapping from identifiers to integer values), and $L$ is a map from labels to indices in $I$. Prior to the execution, statements and expression indices are placed into $I$ by performing a pre-order traversal of the program's abstract syntax tree (the first statement is at index 0). Also, $L$ is initialized to map each label to the appropriate index in $I$. We also use the following operations: (1) map lookup $\_(val)$, and (2) map update $\_[val/\_]$.

Figure 2.3 shows the operational semantics of the language. For simplicity,

$$\text{ASSIGN\_EXP: } \frac{\langle pc+1, I, M, L\rangle exp \Rightarrow \langle pc', I', M, L\rangle v}{\langle pc, I, M, L\rangle id = exp \Rightarrow \langle pc', I', M, L\rangle id = v}$$

$$\text{ASSIGN\_VAL: } \frac{M' = M[v/id]}{\langle pc, I, M, L\rangle id = v \Rightarrow \langle pc+1, I, M', L\rangle I(pc+1)}$$

$$\text{GOTO: } \frac{}{\langle pc, I, M, L\rangle \text{goto } l \Rightarrow \langle L(l), I, M, L\rangle I(L(l))}$$

$$\text{IF\_EXP: } \frac{\langle pc+1, I, M, L\rangle exp \Rightarrow \langle pc', I', M, L\rangle v}{\langle pc, I, M, L\rangle \text{if}(exp)\ N \Rightarrow \langle pc', I', M, L\rangle \text{if}(v)\ N}$$

$$\text{IF\_VAL: } \frac{pc' = (v == 0)\ ?\ pc+1 : L(l)}{\langle pc, I, M, L\rangle \text{if}(v)\ l \Rightarrow \langle pc', I', M, L\rangle I(pc')}$$

$$\text{C\_HOLE: } \frac{val = alt(range(MIN, MAX)) \qquad I' = I[val/pc]}{\langle pc, I, M, L\rangle [\![c]\!] \Rightarrow \langle pc, I', M, L\rangle val}$$

$$\text{V\_HOLE: } \frac{id = alt(identifiers(M)) \qquad I' = I[id/pc]}{\langle pc, I, M, L\rangle [\![v]\!] \Rightarrow \langle pc, I', M, L\rangle id}$$

$$\text{E\_HOLE: } \frac{e = alt([e_1, e_2, ...., e_n]) \qquad I' = I[e/pc]}{\langle pc, I, M, L\rangle [\![e_1, e_2, ..., e_n]\!] \Rightarrow \langle pc, I', M, L\rangle e}$$

Figure 2.3: Semantics for the simple language from Figure 2.2.

the rules do not include error handling. The assignment statement simply updates the value of a variable in memory. The goto statement unconditionally jumps to the statement with the specified label. The conditional statement evaluates the expression and then jumps if the expression evaluates to true ($val \neq 0$). We do not show the rules for computing basic expressions, as we assume the same semantics as in the C programming language. The halt statement terminates the execution.

**Semantics (template language)**. We define several utility functions for the template language: (1) `identifiers`$(M)$ - returns a list of available variable names in $M$ at the point of an invocation, (2) `range`$(x, y)$ - returns a list of integers between $x$ and $y$, and (3) `alt`([...]) - takes a sequence as input and outputs one of its elements.

A hole for an integer literal (C_HOLE) evaluates to an integer literal and rewrites itself to that literal. A hole for a variable (V_HOLE) evaluates to an available identifier and rewrites itself to that identifier. Finally, a hole for an expression (E_HOLE) evaluates to one of the given expressions (and rewrites itself to that expression). Note that the rewrite rules are such that the entire hole is replaced with the choice of a concrete expression upon execution, so the hole no longer exists after the first evaluation, ensuring that each hole evaluates only to a single expression per execution. The program in $I$ at the end of execution is the generated program in the same language as the original template.

**Filling a hole**. Given a list of candidates for a hole, we need to explore different candidates every time we execute the program, which would in turn rewrite the template into new concrete programs that we can later use for testing compilers. While one can try and systematically explore all the possible candidates, the search space can be incredibly large (e.g., for ⟦`c`⟧ the range of possible integers go from `MIN` to `MAX`), especially when considering combinations of candidates chosen across all holes in the program.

For this work, we choose candidates for a hole randomly. Random exploration has been found effective in prior work [84, 103, 139, 143]. We keep re-executing the template to rewrite into concrete programs up until we reach a specified limit for number of generated programs. Note that each execution of a template is independent of other executions, i.e., any modifications to the template during one run is *not* observable in another run.

**Example**. Consider the following example in our language: `s1 = ⟦c⟧; s2 = ⟦c⟧; if (⟦v⟧ < ⟦v⟧) l9; l9:  halt;`. Executing this template once might generate: `s1 = 45350238; s2 = 681339300; if (s1 < s2) l9; l9:  halt;`. Another execution can lead to a different generated program: `s1 = 125652422; s2 = 23297; if (s2 < s2) l9; l9:  halt;`.

### 2.3.2 JAttack Implementation for Java

We implement the semantics of JAttack for the host Java programming language. To support the concept of holes while integrating it into Java, we introduce a set of API methods that construct holes.

Figure 2.4 shows a subset of the API we provide. This API represents a DSL

```
BoolVal boolVal();
IntVal intVal(int min, int max);
IntVal intVal();
BoolId boolId(String... names);
IntId intId(String... names);
...
<T> RefId<T> refId(Class<T> type, String... names);
<T extends Number> BAriExp<T> arithmetic(Exp<T> left, Exp<T> right, Op... ops);
<T extends Number> RelExp<T> relation(Exp<T> left, Exp<T> right, Op... ops);
LogExp logic(Exp<Boolean> left, Exp<Boolean> right, Op... ops);
...
<T> ExprStmt exprStmt(Exp<T> exp);
IfStmt ifStmt(Exp<Boolean> cond, Stmt thenStmt, Stmt elseStmt);
WhileStmt whileStmt(Exp<Boolean> cond, Stmt body);
...
<T> Exp<T> alt(Exp<T>... exps);
Stmt alt(Stmt... stmts);
```

Figure 2.4: API for writing holes; a call to any of the methods in the API instantiates an eAST node.

that maps to our simple imperative language from Figure 2.2 and 2.3. All methods in the API return an instance of a node rooting an eAST. An `Exp<Integer>` node corresponds to an expression (evaluating to integer due to Java typing, so `Exp<Boolean>` is the same for boolean type). An `IntVal` (extends `Exp<Integer>`) node and an `IntId` (extends `Exp<Integer>`) node correspond to an integer literal and variable, respectively. We define `BoolVal` and `BoolId` to correspond to the boolean type for Java. `BAriExp<Integer>` (extends `Exp<Integer>`) is for binary arithmetic expressions, while `RelExp` (extends `Exp<Boolean>`) and `LogExp` (extends `Exp<Boolean>`) are for relational and logical expressions. These nodes are therefore placeholders for the actual, concrete expressions to be generated at runtime, so they represent holes to be filled. For example, an `IntVal` node created using the method `intVal` represents $[\![c]\!]$, a hole that can evaluate to any integer from `Integer.MIN_VALUE` to `Integer.MAX_VALUE`. We also provide an `intVal` API that can specify the range of integer values, and an `intId` API that can enumerate available variables (at any point) by analyzing bytecode, when no variable is specified.

While specific API methods can create corresponding nodes, e.g., `arithmetic` for `BAriExp`, we also provide method `alt` that can choose from a provided list of `Exp<Integer>` or `Exp<Boolean>` nodes, which corresponds to the semantics for an expression hole (E_HOLE) in our simple imperative language.

Although we illustrate the implementation of JATTACK using `int` and `boolean`, we support any other primitive type, e.g., `long` and `double`, or any reference type. For instance, `refId` can enumerate available variables (at any point) with type `T` that can be specified using argument `Class<T> type`, e.g., `refId(String.class)` returns a `RefId<String>` node corresponding to any available `String` variables at this execution point. We provide API methods to create statements as well, such as `exprStmt`, `ifStmt` and `whileStmt`. Furthermore, one can extend our implementation to include more language constructs in Java, as along as they can be represented in an eAST.

As an alternative, we originally designed our API to use a list of concrete Java expressions to choose from, e.g., `alt(i++, j++)`. However, these expressions would get executed and result in side-effects, and the final execution would not match executing the corresponding generated program with the concrete expressions substituting for the hole, so we abandoned that direction.

Instead, when using eAST nodes, we do not actually generate an expression to fill a hole until the `eval` method is invoked on the node, e.g., `intVal().eval()`. Only after calling `eval` does a concrete node get generated for that hole. Once generated, the node is interpreted to compute the result of the expression. Furthermore, all subsequent calls to the same API method (from the same location) will always return the same node. For our Java implementation, we define a hole to be where the developer calls `eval` for a built eAST. The eAST constructed for an API call represents a range of candidates to fill the hole. As an example, consider the hole specified by the `logic` call (lines 10-11 in Figure 2.1a). Executing the `logic` method returns a root node of an eAST, illustrated in Figure 2.5. Candidates for the hole

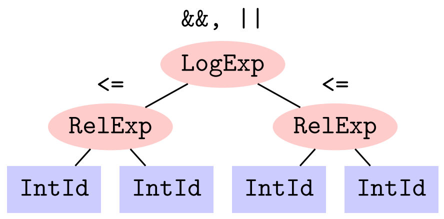

Figure 2.5: eAST corresponding to `logic` hole from Figure 2.1a.

are obtained by recursively obtaining candidates for nodes in subtrees and combining them together.

In this example, the `RelExp` nodes would result in candidates that combine choice of integer variables combined with the specified operators (just `LE` in the example); the top-level `LogExp` node would use the returned candidates and combine with the specified `AND` or `OR` to create the final candidates. This eAST structure corresponds to an expression hole in our imperative language, namely the following:

$[\![ [\![v]\!]$ `<=` $[\![v]\!]$ `&&` $[\![v]\!]$ `<=` $[\![v]\!]$`,` $[\![v]\!]$ `<=` $[\![v]\!]$ `||` $[\![v]\!]$ `<=` $[\![v]\!] ]\!]$.

Unlike our simple imperative language, our API provides syntactic sugars to describe a large set of similar candidates without having to enumerate all of them by specifying multiple operators at once (see `Op... ops` in Figure 2.4).

### 2.3.3 Generation Procedure

Figure 2.6 shows the overall algorithm for JAttack's `Generate` function that executes a template repeatedly to generate concrete program instances. The input to `Generate` is a template $T$ and the number of programs to generate $M$. The output is a set of generated programs $G$.

Function `Generate` starts by initializing the empty set of generated programs $G$ and then capturing the initial global state of the template $T$ into variable $S$ (line 6). We initially supported capturing static fields with primitive types as the global state; later we extended JAttack to support reference types [12, 13] (Section 3.3.3). We capture the global state to be used later when generating programs, ensuring the

```
 1: Input: template program $T$
 2: Input: number of programs to generate $M$
 3: Output: set of generated programs $G$
 4: function Generate($T$, $M$)
 5:     $G \leftarrow \emptyset$
 6:     $S \leftarrow$ CaptureGlobalState($T$)
 7:     $entryMeth \leftarrow$ FindEntryMethod($T$)
 8:     $num \leftarrow$ CountHoles($T$)
 9:     repeat
10:         resetGlobalState($S$)
11:         $P \leftarrow$ RunTemplate($T$, $entryMeth$, $num$)
12:         $G \leftarrow G \cup \{P\}$
13:     until $|G| = M$
14:     return $G$
15: Input: template program $T$
16: Input: entry method $entryMeth$
17: Input: number of holes $num$
18: Output: generated program $P$
19: function RunTemplate($T$, $entryMeth$, $num$)
20:     $H \leftarrow \{\}$
21:     $seenStates \leftarrow \emptyset$
22:     $Q \leftarrow T$
23:     for $i \leftarrow 1$ to $MAX\_NUM\_ITERATIONS$ do
24:         $H' \leftarrow$ ExecEntryMethod($entryMeth$, $Q$)
25:         $H \leftarrow H \cup H'$
26:         if $|H| = num$ then break
27:         $R \leftarrow$ CaptureGlobalState($Q$)
28:         if $R \in seenStates$ then break
29:         $seenStates \leftarrow seenStates \cup \{R\}$
30:         $Q \leftarrow$ HotFill($Q$, $H$)
31:         $Q \leftarrow$ RemoveDeadCode($Q$, $H$)
32:     return ApplyFilledHoles($T$, $H$)
```

Figure 2.6: Generation algorithm.

generation of each (out of $M$) program is done from the clean state. (We use the Java reflection mechanism to capture the state.) Additionally, `Generate` finds the template entry method (line 7), which is the entry point for executing $T$ (in our Java implementation, this is the method annotated with `@Entry`), and also counts the total number of holes that should be filled in the template (line 8).

Next, `Generate` repeatedly calls `RunTemplate`, which executes the template, resulting in a generated program that is added to $G$. Assume that a template always terminates, which can be guaranteed through carefully specifying the search space for the holes in conditions, the overall loop in `Generate` ends when the number of uniquely generated programs has reached the maximum necessary number $M$. We set a timeout, for `RunTemplate`, as it might not be feasible to generate the specified number of unique programs.

Before calling `RunTemplate` in each iteration, `Generate` sets the global state to be the same as the initial global state $S$ (line 10). Setting the initial state to $S$ ensures that subsequent runs of `RunTemplate` always start the generation process, which executes the same template entry method, in the clean state.

**Example**. Consider the template from Figure 2.1a. The template has a static variable `s1` that is modified (line 7). Subsequent executions should make sure `s1` starts at 0 again, otherwise they would not be starting at the same state and would not generate programs that are even possible.

Function `RunTemplate` is responsible for generating a single concrete program from the given template $T$. First, it initializes $H$ as an empty mapping from holes to their filled expressions (line 20). `RunTemplate` then sets an intermediate program $Q$ to be the input template program $T$ to start with (line 22), and then it repeatedly executes the entry method *entryMeth* on $Q$ (line 24). The `ExecEntryMethod` returns a mapping $H'$ of holes it filled to the actual expressions.

**Example**. In Figure 2.1a, executing the first hole on line 7 would result in a mapping of that hole to concrete value 45350238 (line 7 in Figure 2.1b).

The overall mapping $H$ gets updated with $H'$. If all holes have been filled, then the loop terminates (line 26 in Figure 2.6). The reason for executing the template entry method *entryMeth* many times is to ensure all holes that can be reached get filled. Eventually, our goal is to execute a corresponding generated entry method up to `MAX_NUM_ITERATIONS` times as to trigger JIT compiler optimizations (Section 2.3.5). Some holes may only be reachable after multiple iterations, so executing just once would not fill those holes.

**Example**. Consider the template from Figure 2.1a. The last hole (line 12) could be skipped in the first run because the condition (line 10-11) is evaluated to `false`. However, the hole could be filled later when static variable `s1` gets updated (line 7), making the condition `true`.

Some choice of candidate for a hole may possibly make another, later hole unreachable, making it dead code. JAttack may fill a hole in a condition, such as for an `if` statement, that always evaluates to `false`, and therefore any holes within the block of these conditional statements cannot be reached. To prevent the execution from `RunTemplate` from continuously executing while being unable to fill those unreachable holes, `RunTemplate` stops after the `MAX_NUM_ITERATIONS` maximum number of iterations. Having unfilled dead-code holes in a generated program is fine because such code should never even be executed within the maximum number of iterations later (and if it is executed, that would indicate a bug in the JIT compiler). `RunTemplate` does stop earlier when all holes are filled (line 26).

Three optimizations are introduced to reduce generation cost (line 27-31) and we describe the optimizations in detail in Section 2.3.4. Note that $Q$ is an intermediate program, and we do not directly return $Q$. As such, we can optimize and make extra changes in $Q$ to speed up generation, and these changes do *not* belong in a final generated program $P$.

The final returned program $P$ is then the original template $T$ with all its holes filled using the mapping $H$ (computed using function `ApplyFilledHoles` in line 32; the details of `ApplyFilledHoles` function are not shown). Essentially, each node corresponding to a filled hole can output the concrete code snippet for the expression it currently holds, and the hole expression in the template $T$ gets replaced with this concrete code snippet. `Generate` then takes the returned program $P$ and adds it to the running set of generated programs $G$. Note that `Generate` will keep calling `RunTemplate` until obtaining a sufficient number of programs (line 13); each time, `Generate` will use the fresh template program $T$, which has no filled holes, as to create a brand new generated program.

### 2.3.4 Optimizations for Generation

We develop three optimizations to speed up the generation process. Note that these optimizations all apply only for a single run of `RunTemplate`, i.e., just a single generated program at a time. These optimizations do *not* impact the generated programs; they only speed up the generation process.

**Early stopping**. We can have an even earlier stopping condition based on the insight that if the global state after execution is the same as an already seen state, then any future run would lead to the same behavior (as a previous execution). Starting execution in the same global state cannot lead to new executions that fill new holes. In `RunTemplate`, we keep track of the seen global states in `seenStates` and check the global state after each execution (line 28 in Figure 2.6). This type of program state hashing has been extensively used in software model checking [67].

**Hot filling**. In our preliminary experiments, we found that executing a template entry method many times is time-consuming, especially compared to executing the generated entry method as part of our evaluation. The extra overhead comes from repeated executions of our Java API methods that build and evaluate eAST nodes. Recall that during generation the filled hole is not rewritten into the concrete expression, but just evaluated to produce the same value as the concrete expression. The filled holes get replaced with the actual code only when the entire template gets translated into a new generated program (line 32 in Figure 2.6). Thus, while our implementation ensures that repeated execution of the same API method returns the same eAST node, invoking the `eval` method to evaluate the node is expensive compared to evaluating the concrete code that replaces the hole in the generated program.

The hot filling optimization replaces the hole at runtime (during program generation) with the concrete expression when the hole is evaluated for the first time, so that execution in the following iterations can use concrete code rather than invoking our Java API methods, including the `eval` method that evaluates the filled hole. In `RunTemplate`, we invoke the method `HotFill` (line 30 in Figure 2.6) on the resulting $Q$ after execution that finds all API calls with set nodes and, using the mapping of holes to expressions $H$, replaces those calls with the concrete expressions. Then, using interfaces provided from package `javax.tools`, e.g., `javax.tools.JavaCompiler`, we implement an in-memory Java compiler, file manager, and associated class loader to dynamically compile $Q$ and then reload this modified template's class, resulting in a new $Q$. The next iteration starts from the new $Q$ as the template (line 24 in Figure 2.6). This technique is conceptually similar to "quickening" optimization implemented in self-optimizing interpreters [19, 66].

**Eager pruning**. In our preliminary experiments, we also noticed a significant number of generated programs with conditional expressions that are trivially false, e.g., `(var1 > var1)`. The body of such conditional statements would never be executed, so it is unnecessary to execute any further to fill holes within statements guarded by that condition. After executing the template entry method and obtaining filled holes in $H$, we invoke function `RemoveDeadCode` to eliminate any such dead code in the program $Q$ (line 31 in Figure 2.6), completely rewriting the body into an empty statement. This technique is conceptually similar to partial evaluation [71]. We leverage a modern constraint solver (Z3 [39]) in our implementation to determine whether any conditional expression is satisfiable or not, eliminating code in case the expression is unsatisfiable. Note that we only temporarily remove the code as a means to speed up generating a single program. The returned generated program does not have any unreachable code removed. Later calls to `RunTemplate` always start with the same template $T$ that has all the code still present.

### 2.3.5 JIT Compiler Testing Procedure

Revealing JIT compiler bugs requires not just programs but also executing those programs many times. For each generated program, we iterate though different JIT compilers. For each JIT compiler, we repeatedly execute the generated entry method, hashing the output of each execution (a generated entry method's return value is always encoded into an integer) into a running total. After executing `MAX_NUM_ITERATIONS` times (the same limit in Figure 2.6), we capture the global state of generated program (values of all static fields) and encode it within a checksum value, adding this to the running total. The final total represents the combination of all the executions: $\sum_{i=1}^{N} \text{hash}(r_i) + \sum_{f \in \text{class}} \text{hash}(v_f)$, where $N$ is `MAX_NUM_ITERATIONS`, $r_i$ is the return value from the entry method at the $i$-th iteration, $f$ is every static field of the class that declares the entry method, $v_f$ is the final value of the static field $f$ after all the iterations, and the hash function computes a deterministic and unique hash code for any primitive value or object.

For a given generated program, we use differential testing [87] to check if the running totals computed from all JIT compilers are the same. Any difference should indicate that the generated program detected a bug within some JIT compiler. However, the program may itself be non-deterministic, i.e., having different outputs when run multiple times on the same JIT compiler, due to random number generators, timestamps, etc. To avoid being misled by non-determinism, when there are differences in output across different JIT compilers, we choose a JIT compiler as a reference point and run the program twice using that same compiler. If the outputs from running on the same JIT compiler differ, then output differences between JIT compilers do not indicate a bug. While this step may potentially miss detecting some bugs, it gives higher guarantees that reported bugs are true bugs.

Besides checking for differences in final running totals, we also report a bug if the execution crashes on some JIT compiler. Executing any unfilled hole (left as the API method call in the generated program) would also trigger a crash, because an unfilled hole should not be reachable. The ultimate output of the entire JIT compiler testing procedure is a subset of generated programs that expose a bug in one of the input JIT compilers.

## 2.4 Experiment Setup

We briefly describe our evaluation setup.

### 2.4.1 Evaluation Subjects

We wrote 13 templates that exercise Java language features. We also studied the available optimizations used in the HotSpot JIT compiler, creating six templates whose basic structure would trigger those optimizations while including holes for JAttack to explore. Finally, we studied existing bug reports for JIT compiler bugs, creating four templates by modifying the programs attached to bug reports to include holes. In our evaluation, we refer to the templates based on our own understanding of Java and the compiler developers' intuition of optimizations using prefix "M". We refer to the templates based on bug reports using prefix "B". Overall, we created 23 templates, with the goal to evaluate the effectiveness of our optimizations.

Although hand-written templates provide unique insights on compilers to be tested, we also evaluate JAttack for automated end-to-end compiler testing. We collect templates automatically from existing Java code. We use 77 open-source Java Maven projects from GitHub to extract templates from their classes. Given a project or a module of a multi-module Maven project, we find classes defined in all ".java" files.

Given a Java class, we first parse all the available methods in the class to detect potential holes. For each statement, we recursively convert each subexpression into the corresponding hole, starting from the leaves of the expression tree. For example, the expression `a + b` would be converted into `arithmetic(intId(), intId()).eval()` (specifying no operator argument means using all valid operators), which matches the expression structure. Note that the final call to `eval` is on the outermost API call, allowing for the greatest space of combination of values that JAttack can explore.

After inserting holes into the Java class, we then scan the class for available static methods, which are the candidate template entry methods. If the static method takes any parameters, we insert additional *parameter methods*, one for each parameter; a parameter method returns a concrete value for the corresponding parameter type upon execution. For primitive values, we leverage JAttack to provide a possible value, e.g., if the parameter is an `int` type we simply use `intVal` to represent an integer value. For non-primitive types, i.e., classes, we search if such classes have default constructors or constructors with primitive arguments that we can simply use to create an instance of that class. If there are no such constructors, we search from other classes for a public static method that returns an instance of the class. If none of the above cases applies, we then use `null` as the concrete value.

The automated approach to extracting templates has been proved to be effective on discovering JIT compiler bugs, which inspired us to develop an overarching framework around JAttack for automated end-to-end Java JIT compiler testing. We describe a systematically developed framework for end-to-end Java JIT testing in Chapter 3.

### 2.4.2 Configuring JAttack

For each template we created ourselves, we configure JAttack to generate 1,000 concrete programs (`M` in Figure 2.6). While generation is fastest when we turn on all three generation optimizations (Section 2.3.4), we also evaluate running generation without any optimization and with each optimization separately, measuring each one's effectiveness. For each of the generated programs, we execute it 100,000 times (`MAX_NUM_ITERATIONS` in Figure 2.6) on different JIT compilers. The JIT compilers we evaluate on are the HotSpot JIT compiler from Oracle JDK with version

Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.8+10-LTS), the HotSpot JIT compiler from OpenJDK 64-Bit Server VM AdoptOpenJDK (build 11.0.8+10), and the OpenJ9 JIT compiler from Eclipse OpenJ9 VM AdoptOpenJDK (build openj9-0.21.0, JRE 11 Linux amd64-64-Bit Compressed References 20200715_697), all based on JDK 11.0.8.

For templates extracted from existing Java projects, we follow the same approach, except we configure JAttack to generate only 10 concrete programs from each template program, because of the large number of templates, and we test only on the latest Oracle JDK, which was Java HotSpot(TM) 64-Bit Server VM (build 16.0.2+7-67) at that time.

In our evaluation, we configure the HotSpot JIT compiler to restrict the specific tiers, L1 and L4, using the option `-XX:TieredStopAtLevel`, in order to test C1 and C2 compilers [94], respectively. We treat each restricted tier configuration for the HotSpot JIT compiler as conceptually a new JIT compiler for use in our JIT compiler testing procedure (Section 2.3.5).

We run all experiments in this chapter on a 64-bit Ubuntu 18.04.1 desktop with an Intel(R) Core(TM) i7-8700 CPU @3.20GHz and 64GB RAM. For all time measurements, we run the evaluation five times and report the average of those times.

## 2.5 Evaluation

We evaluate JAttack by asking:

**RQ1**: How efficient is JAttack at generating programs and executing those generated programs with different JIT compilers?

**RQ2**: How well can JAttack be used for *automated* compiler testing via extracted templates from a large number of existing Java programs, and how does it compare with the state-of-the-art automated JIT compiler testing?

**RQ3**: What critical bugs does JAttack detect in JIT compilers?

We address RQ1 as to better understand how efficient JAttack is at generating programs from templates, as well as the impact of our optimizations for generation, and to understand the efficiency of JAttack's testing procedure. We address RQ2 to understand how well JAttack can be used for automated compiler testing and compare the effectiveness with tools used in industry. We address RQ3 to understand the bugs that we discovered. JAttack and all JIT compiler bugs we detected, including associated templates and generated programs, are available at `https://github.com/EngineeringSoftware/jattack`.

### 2.5.1 Performance and Optimizations

Table 2.1 shows the time for JAttack to generate 1,000 programs for each of our manually created 23 templates. The different columns show the total time to generate all 1,000 programs when using different generation optimizations (Section 2.3.4). Namely, Non-optimized means no optimizations, Early Stopping means using only early stopping, Hot Filling means using only hot filling, and Eager Pruningmeans using only eager pruning. The final column for Fully Optimized is the time when using all optimizations. In addition to time, for each optimization column, we also show the percentage of time reduced relative to Non-optimized time (the higher the reduction the better). The final row shows the sum of generation time across all templates and the overall reduction over this total time.

Without any optimizations, the total time for generation across all templates (essentially 1,000 $*$ 23 $=$ 23,000 programs total) is over two days. We find that the overall time drops tremendously after the optimizations are in place. When all optimizations are enabled (Fully Optimized), the overall time to generate all programs for all templates is around 20 minutes, which is a 99.50% reduction over the time it takes to generate all programs without any optimization.

Table 2.1: Time ((dd:)hh:mm:ss) and relative reduction to Non-optimized (%) to generate 1,000 programs in various configurations.

| Templates | Non-optimized | Early Stopping | | Hot Filling | | Eager Pruning | | Fully Optimized | |
|---|---|---|---|---|---|---|---|---|---|
| | Time | Time | Reduction | Time | Reduction | Time | Reduction | Time | Reduction |
| B1 | 05:00:36 | 00:14 | 99.92 | 01:31 | 99.50 | 02:55:07 | 41.74 | 00:56 | 99.69 |
| B2 | 12:51:25 | 00:14 | 99.97 | 04:14 | 99.45 | 12:53:50 | -0.31 | 01:03 | 99.86 |
| B3 | 01:19 | 00:12 | 84.33 | 00:33 | 58.57 | 02:01 | -53.59 | 00:13 | 84.08 |
| B4 | 01:43 | 01:41 | 2.37 | 01:41 | 2.03 | 01:41 | 2.45 | 01:43 | 0.29 |
| M1 | 11:32 | 00:10 | 98.60 | 00:42 | 93.96 | 10:17 | 10.82 | 00:33 | 95.22 |
| M2 | 12:57 | 00:12 | 98.49 | 00:44 | 94.29 | 14:40 | -13.22 | 00:34 | 95.59 |
| M3 | 11:00 | 02:19 | 78.90 | 01:05 | 90.14 | 13:04 | -18.88 | 00:56 | 91.49 |
| M4 | 19:44 | 04:31 | 77.12 | 00:28 | 97.60 | 20:35 | -4.26 | 00:31 | 97.39 |
| M5 | 05:14:14 | 46:01 | 85.36 | 01:04 | 99.66 | 04:20:07 | 17.22 | 00:46 | 99.76 |
| M6 | 03:05 | 00:09 | 95.08 | 00:45 | 75.79 | 04:21 | -41.40 | 00:38 | 79.43 |
| M7 | 05:24 | 05:31 | -2.40 | 00:38 | 88.19 | 05:41 | -5.39 | 00:49 | 84.74 |
| M8 | 19:19 | 12:26 | 35.62 | 01:08 | 94.14 | 21:10 | -9.62 | 01:16 | 93.43 |
| M9 | 02:25 | 00:14 | 90.42 | 00:28 | 80.59 | 02:49 | -16.18 | 00:20 | 85.98 |
| M10 | 09:07 | 02:55 | 67.96 | 00:41 | 92.50 | 10:32 | -15.63 | 00:39 | 92.90 |
| M11 | 10:58 | 00:10 | 98.48 | 02:15 | 79.44 | 08:35 | 21.79 | 01:02 | 90.62 |
| M12 | 04:23 | 04:40 | -6.38 | 00:36 | 86.51 | 05:24 | -22.88 | 00:43 | 83.64 |
| M13 | 11:35:19 | 11:26:37 | 1.25 | 01:06 | 99.84 | 00:57 | 99.86 | 00:51 | 99.88 |
| M14 | 11:40:46 | 05:43:54 | 50.93 | 01:38 | 99.77 | 11:58:25 | -2.52 | 01:17 | 99.82 |
| M15 | 03:55:35 | 00:09 | 99.94 | 00:45 | 99.68 | 02:49:16 | 28.15 | 00:14 | 99.90 |
| M16 | 07:38:42 | 07:47:57 | -2.02 | 01:02 | 99.77 | 07:57:36 | -4.12 | 01:09 | 99.75 |
| M17 | 10:38:24 | 11:28:05 | -7.78 | 03:12 | 99.50 | 10:51:56 | -2.12 | 03:32 | 99.45 |
| M18 | 04:57 | 00:09 | 96.81 | 00:41 | 86.07 | 05:43 | -15.45 | 00:32 | 89.08 |
| M19 | 05:47 | 05:53 | -1.82 | 01:01 | 82.34 | 03:51 | 33.41 | 01:02 | 82.16 |
| Σ | 2:22:38:42 | 1:13:54:24 | 46.34 | 27:59 | 99.34 | 2:07:57:38 | 20.79 | 21:20 | 99.50 |

Breaking down the effectiveness of our optimizations even further, we find that the hot filling optimization is in general the most effective, with hot filling reducing the generation time by 99.34% versus 46.34% for early stopping and 20.79% for eager pruning. Furthermore, we also see that early stopping and eager pruning have cases where they result in taking more time to generate programs than without any optimization (the negative percentages in the table), which suggests the extra checks required by early stopping and the time to invoke Z3 to solve constraints end up introducing more overhead than actually helping. (We did not set a timeout for Z3, because we did not observe Z3 getting stuck; however, setting a timeout could impact the performance of the eager pruning optimization.) We see just one case for hot filling (and ultimately for when all optimizations are on) where there is little reduction in time. However, this one case (B4) takes very little time even without any optimizations, and the difference in time is seemingly just noise. Ultimately, all optimizations do help overall, with the reduction in time when using all optimizations still higher than each individually.

We also measure the time to execute the generated programs from each of the 23 manually created templates. The total time across all generated programs is around two hours on L4 and around two and a half hours on L1.

### 2.5.2 Template Extraction

As is shown in Table 2.2 and 2.3, we extract 5,419 templates from 16,309 methods in 15,325 classes, resulting in 50,609 generated programs. Recall that we let JAttack generate 10 programs from every template (Section 2.4), but not every template includes sufficient number of holes from which 10 programs can be generated (JAttack only explores the reachable holes), which is why the total number of generated programs is less than $10 * 5{,}419 = 54{,}190$. We found 137 out of 50,609 generated programs failed during our JIT compiler testing procedure. We inspected all these 137 programs and discovered five unique bugs (Section 2.5.3).

In addition, we compare JAttack against an existing automated compiler testing tool, Java* Fuzzer [9], which is a fuzzer tool that Oracle has been using daily for years and has been successful at detecting bugs in the HotSpot JIT compiler. Guided by grammar rules and predefined heuristics on program structures, Java* Fuzzer generates hundreds of thousands of small, random Java programs as tests, and it then performs differential testing between a JVM under test and a reference JVM. In contrast, JAttack is primarily developed for developers to embed their knowledge into program generation by specifying holes in templates with automated template extraction from existing Java programs. Although JAttack and Java* Fuzzer have similar intentions, they work quite differently, which is why the com-

Table 2.2: Results of template program extraction from existing Java projects, including the number of templates, the total number of generated programs and the number of generated programs that failed JIT compiler testing per project.

| **Project** | **# Templates** | **# Programs** | **# Failures** |
|---|---|---|---|
| checkstyle | 194 | 1,219 | 23 |
| commons-codec | 145 | 1,339 | 45 |
| commons-compress | 140 | 1,350 | 0 |
| commons-configuration | 8 | 58 | 0 |
| gson | 11 | 98 | 0 |
| jfreechart | 158 | 1,482 | 1 |
| commons-jxpath | 37 | 355 | 0 |
| commons-lang | 733 | 7,036 | 0 |
| libgdx | 359 | 3,466 | 0 |
| commons-math | 564 | 5,484 | 9 |
| Openfire | 137 | 1,030 | 0 |
| vectorz | 398 | 3,622 | 0 |
| zxing | 240 | 2,090 | 2 |
| bootique | 6 | 60 | 0 |
| docker-maven-plugin | 88 | 861 | 0 |
| easy-random | 5 | 50 | 0 |
| find-sec-bugs | 207 | 2,070 | 0 |
| game-server | 112 | 1,075 | 2 |
| gravitee-gateway | 1 | 10 | 0 |
| Jupiter | 61 | 455 | 0 |
| mango | 38 | 347 | 0 |
| mysql_perf_analyzer | 1 | 10 | 0 |
| spring-cloud-config | 3 | 16 | 0 |
| streamex | 6 | 60 | 0 |
| ta4j | 22 | 220 | 0 |
| whatsmars | 58 | 575 | 0 |
| aviatorscript | 39 | 354 | 0 |
| bytecode-viewer | 52 | 442 | 0 |
| Chronicle-Queue | 37 | 362 | 0 |
| docker-java | 4 | 40 | 0 |
| flyway | 47 | 428 | 0 |
| javacpp | 10 | 100 | 0 |
| jsonschema2pojo | 1 | 10 | 0 |
| jsoup | 79 | 790 | 0 |
| JSqlParser | 12 | 120 | 0 |
| metrics | 0 | 0 | 0 |
| Recaf | 187 | 1,816 | 2 |
| ripme | 29 | 238 | 38 |
| spring-cloud-gateway | 16 | 160 | 0 |
| spring-data-jpa | 5 | 44 | 0 |
| WePush | 26 | 251 | 0 |
| byte-buddy | 36 | 359 | 0 |
| commons-bcel | 57 | 570 | 0 |
| commons-digester | 2 | 20 | 0 |
| dnsjava | 85 | 831 | 0 |
| easy-batch | 4 | 40 | 0 |
| equalsverifier | 4 | 10 | 0 |
| graphql-spqr | 7 | 70 | 0 |

Continued

Table 2.3: Table 2.2 continued.

| **Project** | **# Templates** | **# Programs** | **# Failures** |
|---|---:|---:|---:|
| james-mime4j | 21 | 204 | 14 |
| javassist | 224 | 2,227 | 0 |
| jsql-injection | 8 | 80 | 0 |
| karaf-cellar | 0 | 0 | 0 |
| lambda | 2 | 20 | 0 |
| mug | 9 | 90 | 0 |
| ormlite-core | 8 | 64 | 0 |
| pebble | 10 | 100 | 0 |
| simple-java-mail | 3 | 13 | 0 |
| spring-cloud-aws | 1 | 10 | 0 |
| spring-cloud-commons | 1 | 10 | 0 |
| spring-data-neo4j | 7 | 70 | 0 |
| spring-data-rest | 1 | 10 | 0 |
| typescript-generator | 4 | 24 | 0 |
| uima-uimafit | 61 | 603 | 0 |
| carina | 41 | 306 | 0 |
| fluo | 22 | 212 | 0 |
| commons-jexl | 36 | 360 | 0 |
| maven-assembly-plugin | 5 | 50 | 0 |
| maven-indexer | 7 | 52 | 0 |
| maven-wagon | 4 | 40 | 0 |
| commons-numbers | 42 | 415 | 0 |
| commons-ognl | 2 | 4 | 0 |
| one-nio | 298 | 2,899 | 0 |
| phoenicis | 1 | 10 | 0 |
| commons-text | 80 | 798 | 1 |
| turbine | 21 | 204 | 0 |
| commons-validator | 22 | 195 | 0 |
| velocity-tools | 7 | 46 | 0 |
| $\Sigma$ | 5,419 | 50,609 | 137 |

parison results should be taken with a grain of salt. We run Java* Fuzzer using the same resources (CPU/RAM) for the same amount of time (which matches the total execution time for JATTACK in Section 2.5.2). We perform differential testing by comparing outputs from executions across different JIT compiler tiers, same as for JATTACK. We also collect code coverage of both the C1 (`src/hotspot/share/c1/`) and C2 (`src/hotspot/share/opto/`) compilers from executing the programs generated by both tools separately. Table 2.4 compares the results of JATTACK and Java* Fuzzer. Java* Fuzzer did not generate any program that would expose a bug in the HotSpot JIT compiler in the given time frame. Also, JATTACK achieves higher code coverage on both C1 and C2.

Table 2.4: Comparison of JAttack and Java* Fuzzer.

| | #Generated | #Timeout | #Failures | Coverage(%) | |
|---|---|---|---|---|---|
| | | | | C1 | C2 |
| JAttack | 50,609 | 1,243 | 137 | 84.3 | 80.3 |
| Java* Fuzzer | 15,931 | 2,336 | 0 | 80.6 | 67.5 |

Table 2.5: Detected bugs in HotSpot; five bugs were previously unknown.

| Bug ID | Type | JDK Versions | Status | CVE | Duplicates | Source of Templates |
|---|---|---|---|---|---|---|
| JDK-8239244 | Diff | 8, 11, 13 | Fixed | CVE-2020-14792 | - | Hand-written |
| JDK-8258981 | Crash | 9, 10, 11, 15, 16 | Fixed | - | JDK-8250609 | Hand-written |
| JDK-8271130 | Crash | 8, 11, 16, 17 | Fixed | CVE-2022-21305 | - | Extracted |
| JDK-8271276 | Crash | 16, 17, 18 | Fixed | - | - | Extracted |
| JDK-8271459 | Diff | 8, 11, 16, 17, 18 | Fixed | - | - | Extracted |
| JDK-8271926 | Crash | 11, 16 | Fixed | - | JDK-8268362 | Extracted |
| JDK-8297730 | Diff | 9, 11, 17, 18, 19, 20, 21 | Fixed | - | - | Extracted |

```java
public class C {
  static int s1;
  static int s2;
  private static void m() {
    int X = 4_194_304, var1 = 0, i = 0;
    s1 = s1 + X;
    while (i++ < 10 && (s1 <= s2 || s1 > X)) {
      s2 = --s1 + s2; var1 += s1 + s2; } }
  public static void main(String[] args) {
    for (int i = 0; i < 100_000; ++i) m();
    System.out.println(s1 + s2); } }
```

Figure 2.7: Bug JDK-8239244.

### 2.5.3 Detected Bugs

Table 2.5 contains all the seven bugs we discovered during experiments in this section. We show (in Figure 2.1b and 2.7–2.13) and describe all the seven bugs.

We discovered two bugs using the templates we wrote ourselves. JDK-8239244 (Figure 2.7), from template M12, showed mismatching outputs on different tiers because C2's range-check elimination leads to incorrect loop executions. The Oracle developers labeled the bug we reported as a CVE, and they fixed the bug in a recent Oracle Critical Patch Update. The Oracle developers also confirmed JDK-8258981

```java
public class C {
  static int s1;
  static int s2;
  public static void m() {
    int[] arr1 = { s1, s2, 1, 2, 0 };
    for (int i = 0; i < arr1.length; ++i)
      if (arr1[3] <= s2 || s2 <= arr1[2])
        arr1[i] &= s1; }
  public static void main(String[] args) {
    for (int i = 0; i < 100_000; ++i) m(); } }
```

Figure 2.8: Bug JDK-8258981.

```java
public class C {
  public static void m() {
      int X = 536_870_908;
      int[] a = new int[X + 1];
      a[X] = 1; }
  public static void main(String[] args) {
    for (int i = 0; i < 1_000; ++i) m(); } }
```

Figure 2.9: Bug JDK-8271130.

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class C {
  public static void m(String s) {
    Pattern pattern = Pattern.compile("");
    Matcher matcher = pattern.matcher(s); }
  public static void main(String[] args) {
    for (int i = 0; i < 10_000; ++i)
      try { m(null); }
      catch (Throwable e) {} } }
```

Figure 2.10: Bug JDK-8271276.

(Figure 2.1b), where a crash occurred from C2, as a P3[2] bug; this bug was discovered in parallel by others and was fixed in JDK 16. Our template that exposed this bug is shown in Figure 2.1a.

Additionally, we discovered five bugs using extracted templates from existing

[2]P3: Major loss of function.

```java
public class C {
  static String m() {
    StringBuilder sb = new StringBuilder(-1);
    return sb.toString(); }
  public static void main(String[] args) {
    int sum = 0;
    for (int i = 0; i < 10_000; ++i)
      try { m(); }
      catch (Throwable e) { sum += 1; }
    System.out.println(sum); } }
```

Figure 2.11: Bug JDK-8271459.

```java
import java.util.Arrays;
public class C {
  private static void m() {
    int[] arr = { 0 };
    int max = -1;
    for (int i : arr) max = max;
    Arrays.copyOf(arr, max); }
  public static void main(String[] args) {
    for (int i = 0; i < 10_000; ++i)
      try { m(); }
      catch (Throwable e) {} } }
```

Figure 2.12: Bug JDK-8271926.

```java
public class C {
  static byte[] m(byte[] arg1) {
    byte[] b = new byte[-1];
    System.arraycopy(arg1, 0, b, 0, arg1.length);
    return b; }
  public static void main(String[] args) {
    int sum = 0;
    for (int i = 0; i < 100_000; ++i) {
      try {
        System.out.println(m(null));
      } catch (Throwable e) {
        if (e instanceof java.lang.NegativeArraySizeException) {
          sum++; } } }
    System.out.println(sum); } }
```

Figure 2.13: Bug JDK-8297730.

Java projects. JDK-8271130 (Figure 2.9) crashed on tiers L1 and L4 because an array store in C1 compiled code writes to an arbitrary location due to index overflow. The Oracle developers labeled the bug as a CVE, and they fixed the bug in another recent Oracle Critical Patch Update. JDK-8271276 (Figure 2.10) was confirmed as a crash bug, with priority P2[3], related to wrong JVM state used for a receiver null check, and it was fixed in JDK 17. JDK-8271459 (Figure 2.11) missed throwing `NegativeArraySizeException` on tier L4 caused by C2 optimizations; the Oracle developers labeled it as a P2 bug and fixed it in JDK 18. JDK-8271926 (Figure 2.12) crashed due to incorrect C2 loop optimizations before calling `Arrays.copyOf` with a negative parameter; this bug was confirmed with priority P3 and was also discovered in parallel by others. The bug was fixed in JDK 18. JDK-8297730 (Figure 2.13) threw incorrect exceptions from array initialization and following `System.arraycopy` caused by incorrect execution of C2 compiled code; the Oracle developers labeled it as a P3 bug and fixed it in JDK 21.

## 2.6 Discussion

In this section, we contrast JAttack's execution-based generation to static generation and describe limitations of JAttack.

### 2.6.1 Execution-Based vs. Static Generation

Recall that JAttack generates programs through an execution-based model (Section 2.3.3) but we could have generated programs statically by processing an entire template and replacing all holes with concrete expressions. Static generation would process the template repeatedly, putting in different concrete expressions per hole to output a new generated program, up to some maximum number. Generating programs statically could be faster, because it would not be executing the program

[3]P2: Crashes, loss of data, severe memory leak.

at any point.

However, the execution-based generation provides a number of advantages over static generation. Execution-based generation (1) knows what exactly would be executed in a generated program after being compiled, i.e., which parts are dead code or which parts are executable (such information can be leveraged to guide the exploration of holes instead of relying on randomness, which we discuss as future work in Chapter 5) and (2) makes it possible to use values available at runtime to construct holes; consider:

```
int m(int[] a) {
  return a[intVal(0, a.length).eval()];
}
```

where the hole would be a random integer between `0` and the length of the array `a`, which depends on the value of `a`, known only when actually executing the template. An execution-based model allows for expressing more complex programs that static generation cannot generate, as it does not have such runtime information.

To compare execution-based and static generation, we create a variant of JAttack that generates programs statically. This variant relies on the same syntax and semantics, but it statically processes the template once to replace all the holes with concrete expressions. Similar to execution-based generation, we construct eASTs for all the holes. Each eAST per hole contains all the choices for the hole, i.e., concrete expressions that can be filled in the hole. For each hole written in the template, we randomly choose one of the concrete expressions to replace the hole, resulting in a generated program. The generated program has every hole filled, unlike for execution-based generation where some holes may remain unfilled if they are not reached during execution.

For this evaluation, we only consider those templates that are hand-written. We use the same configuration (1,000 programs for each template) for our static variant of JAttack as to allow for proper comparison against the execution-based model. The total generation time with static generation is around three minutes, which is shorter than execution-based generation (around 20 minutes), but they both generate a large number of programs, and executing all generated programs for both approaches still takes almost four and a half hours. As such, generation time is practically negligible compared against the differential testing part of JAttack. Furthermore, since static generation fills every hole in the template, some generated programs could be syntactically different from each other, but their differences are only for expressions in the unreachable holes, so essentially the same code would be executed. Execution-based generation would skip unreachable holes, ensuring every generated program is not only syntactically different but also executed differently. We collected reachability of the filled holes when executing the generated programs. 78.44% of filled holes are reached during execution of generated programs from static generation while execution-based generation guarantees 100.00% reachability. In terms of detected bugs, compared against execution-based generation, statically generated programs detected only JDK-8239244 and missed detecting JDK-8258981.

### 2.6.2 Limitations

There are two main reasons why a relatively small number (5,419) of templates are extracted from a relatively large number (15,325) of classes in existing projects. First, JAttack initially supported only static Java methods as template entry methods. One way to support instance methods as template entry methods is to use Randoop [101] to create receiver objects and inputs for instance methods. Second, we use a different name for the extracted template class from the original class, which sometimes made the template not pass Java type-checking due to circular dependencies between the template class and other classes. To solve this issue, we can edit the original class in place instead of creating a renamed template class. We have extended JAttack to address both limitations in the overarching framework LeJit (see Chapter 3).

JAttack requires re-executing programs many times just to trigger JIT compiler optimizations for testing. We considered the option `-XX:CompileThreshold` that controls the number of interpreted method invocations before optimization. We also considered the option `-XX:Tier4InvocationThreshold` that controls the minimum number of method invocations before transitioning to L4. However, we found these other options also have a big effect on when JIT compiler optimizations occur, so just using these options would not truly reflect actual JIT usage, similar to just enabling C2 from the beginning [61].

## 2.7 Conclusion

We presented JAttack, a framework that enables template-based testing for compilers. Using JAttack, compiler developers can write templates in the same language as the compiler they are testing (Java), enabling them to leverage their domain knowledge to set up a code structure likely to lead to compiler optimizations while leaving holes representing expressions they want explored. JAttack executes templates, exploring possible expressions for holes and filling them in, generating programs to later be compiled using various compilers. To speed up the generation process, we introduced three optimizations that reduced overall generation time by 99.50% in our experiments. Using 23 templates created on our own and 5,419 templates extracted from existing Java projects, JAttack found seven critical (P3 or higher) bugs in the HotSpot JIT compiler, all of which were confirmed and fixed by Oracle developers. Five of them were previously unknown, including two unknown CVEs. JAttack blends the power of developers insights, who are providing templates, and random testing to detect JIT compiler bugs.

To better automate end-to-end Java JIT compiler testing using JAttack and provide templates automatically, we develop an overarching framework around JAttack, which also addresses some limitations of JAttack.

# Chapter 3: Java JIT Compiler Testing with Template Extraction

In this chapter, we present LeJit, a unified automated testing framework around JAttack (Chapter 2), which streamlines creation of templates from existing Java code and testing of Java JIT compilers. LeJit automatically generates template programs from existing Java code by converting expressions to holes, as well as generating necessary glue code (i.e., code that generates instances of non-primitive types) to make generated templates executable. We have successfully used LeJit to test a range of popular Java JIT compilers, revealing five bugs in HotSpot, nine bugs in OpenJ9, and one bug in GraalVM. All of these bugs have been confirmed by Oracle and IBM developers, and 11 of these bugs were previously unknown, including two CVEs. Our comparison with several existing approaches shows that LeJit is complementary to them and is a powerful technique for ensuring Java JIT compiler correctness. [1]

## 3.1 Introduction

Recall that JAttack takes templates as input and each template is a valid Java program with holes, and each hole is written in an embedded domain-specific language that specifies the set of expressions that can potentially fill the hole. JAttack generates programs by fuzzing holes during the execution of a given template. The advantage of this technique is that developers have full control of the space that should be tested and the way programs should be modified. At the same time, JAttack requires substantial developers' engagement, as both template program design and hole values are written manually.

[1]Parts of this chapter are published at FSE 2024 [148]. I led the design, implementation, and evaluation of the system, as well as analyzing the data and writing the paper.

To automate Java JIT compiler testing by using JAttack and provide templates automatically, we present another framework built around JAttack, dubbed LeJit, for automatically generating template programs from existing code. LeJit generates templates by rewriting existing expressions to holes, as well as generating necessary glue code (e.g., code that creates instances of non-primitive types on which methods can be invoked) to make those templates executable. Execution of generated templates, which randomly fills the holes, creates concrete programs that are used as inputs for Java JIT compiler testing. As a result, LeJit sits in between mutation-based techniques and template-based techniques. Unlike existing template-based techniques, templates are automatically extracted from any existing code. Unlike mutation-based techniques, each hole has its own set of values and can be filled dynamically (rather than statically), and multiple holes are filled simultaneously during the execution of the template. Subsequently, LeJit is in a way similar to higher order mutation [70], but holes are filled dynamically by executing templates.

LeJit is designed to enable generation of a program template from any existing method. One of the key challenges was to enable templates for methods that accept instances of complex types as arguments, including an instance on which an instance method is to be invoked. Our key insight in this direction is to capture instances of various types during testing of methods from which templates are to be extracted; tests can be either existing hand-written tests or automatically-generated tests (e.g., using Randoop).

Unlike several existing tools for testing Java runtime environments [57], LeJit generates source code rather than bytecode. Some advantages of focusing on source code rather than bytecode include: 1) eliminating the need to worry about invalid classfiles, as those obtained from source files always pass the early check of the class format performed by bytecode verifiers, allowing for "deeper" testing; 2) simplifying every step during the bug reporting phase: a bug reported as a source code snippet instead of bytecode is easier to understand, minimize, and report, and it also facilitates confirmation, fixing, and integrating in test suites by compiler developers;

3) decreasing the likelihood of revealed bugs resulting in false positives, since these programs result in valid bytecode generated via a Java compiler as opposed to random sequences of bytecode instructions.

We used LeJit to test several JIT compilers: Oracle HotSpot, IBM OpenJ9, and Oracle GraalVM. We used differential testing [87] to detect crash and inconsistency between JIT compilers. We extracted templates from ten open-source Java projects available on GitHub, although our technique can extract templates from any other code. Our runs discovered five bugs in HotSpot, nine bugs in OpenJ9, and one bug in GraalVM; 11 out of the 15 bugs were previously unknown, including two CVEs. All bugs have been confirmed by compiler developers.

We further compared LeJit with JITfuzz [136] and JavaTailor [153], the state-of-the-art testing tools for Java JIT compilers and JVM, respectively. Our experiments show that LeJit increased code coverage of C1 compiler by 8.0% and C2 compiler by 8.2% [94] compared to JITfuzz, and increased by 3.3% and 4.0% compared to JavaTailor, when testing OpenJDK HotSpot. Additionally, using JITfuzz and JavaTailor we have not discovered any of the bugs found by LeJit.

The main contributions of LeJit include:

- **Framework**. We designed and implemented a framework for extracting templates from existing code by converting expressions into holes and capturing instances of complex types during test execution. Captured instances enable execution of templates that produce concrete programs used as inputs for compiler testing.
- **Implementation**. We have implemented LeJit for Java and built it as a unified automated testing framework around JAttack. We have also developed several variants of LeJit to help us understand the benefits of templates and captured instances used for arguments.
- **Evaluation**. We have performed extensive evaluation of LeJit. We have extracted 143,195 templates from ten open-source Java projects on GitHub. We

then used JAttack to generate 886,178 concrete programs. We have used the generated programs to test three compilers – Oracle HotSpot, IBM OpenJ9, and Oracle GraalVM. Additionally, we compare LeJit with JITfuzz and JavaTailor the state-of-the-art tools for testing Java runtime environments.

- **Analysis**. We performed an in-depth analysis of templates and generated programs to understand how the presence of various Java language features, e.g., arrays, conditional statements, loops, etc., affect LeJit's bug detection capabilities. We also studied the impact of various types of templates on the result, and we find types of holes that play an important role in bug detection.
- **Results**. Our results show the effectiveness of LeJit. We have discovered 15 bugs, including five bugs in HotSpot, nine bugs in OpenJ9, and one bug in GraalVM; 11 of the bugs are previously unknown, including two CVEs. All bugs have been confirmed by compiler developers. Our results also show that LeJit is complementary to the state-of-the-art techniques, which did not discover any of the bugs found by LeJit.

LeJit is available at `https://github.com/EngineeringSoftware/lejit`.

## 3.2 Example

We demonstrate the capabilities of LeJit, using an example program that involves a bug we detected in the OpenJ9 JIT compiler. Figure 3.1 shows a snippet of the example program.

LeJit extracts a template from this program by replacing expressions with holes, as shown in Figure 3.2. A hole is a placeholder to be filled with concrete expressions during program generation. Each hole is expressed as an API call, which defines the type and range of values that can be used to fill the hole, e.g., the first hole `refId(String.class)` (line 10) in the template represents any available variable with type `String` at this execution point [144]. (See Section 2.3.1 and 2.3.2 for the

```
 1 package org.apache.commons.text;...
 2 public class StrBuilder implements ... { ...
 3   static final int CAPACITY = 32;
 4   char[] buffer;
 5   private int size;
 6   private String newLine, nullText;
 7   public StrBuilder(final String str) {
 8       if (str❶ == null) { buffer = new char[CAPACITY❷]; } ... }
 9   public StrBuilder trim() {
10     if (size == 0❸) { return this; }
11     int len = size❹;
12     final char[] buf = buffer❺;
13     int pos = 0❻;
14     while (pos < len && buf[pos] <= ' '❼) { pos++; }
15     while (pos < len && buf[len - 1] <= ' '❽) { len--; } ...
16     return this; } }
```

Figure 3.1: An existing program from the text project [126] used as a source for template extraction.

details on holes and API calls.)

There are eight holes displayed in the template, two in the constructor and six in the method `trim`. Each hole, labeled with a circled number, is converted from the expression in the original program with the same number. For example, the first hole `refId(String.class)` (line 10) in Figure 3.2 is converted from the local variable `str` (line 8) in Figure 3.1. The next hole `intId()` (line 11) in Figure 3.2, which represents any available `int` variable, is converted from the `int` field `CAPACITY` (line 8) in Figure 3.1. The third hole (line 15) in Figure 3.2, represents a relational expression that connects an integer variable and an integer literal (between `Integer.MIN_VALUE` and `Integer.MAX_VALUE`) using a relational operator (`<`, `<=`, `>`, `>=`, `==`, `!=`). This hole is converted from the `if` condition `size == 0` (line 10) in Figure 3.1. Similarly, the next three holes: an integer variable hole (line 16), a char array variable hole (line 17), and an integer literal hole (line 18) in Figure 3.2, are converted from `size` (line 11), `buffer` (line 12), and `0` (line 13) in Figure 3.1, respectively. The last two holes are converted from the `while` condition `pos < len && buf[pos] <= ' '` and `pos < len && buf[len - 1] <= ' '`, respectively. The hole 7 (line 22)

```
package org.apache.commons.text;...
import jattack.annotation.*;
import static jattack.Boom.*;
public class StrBuilder implements ... { ...
  static final int CAPACITY = 32;
  char[] buffer;
  private int size;
  private String newLine, nullText;
  public StrBuilder(final String str) {
    if (refId(String.class).eval()❶ == null) {
      buffer = new char[intId().eval()❷]; } ... }

  @Entry
  public StrBuilder trim() {
    if (relation(intId(), intVal()).eval()❸) { return this; }
    int len = intId().eval()❹;
    final char[] buf = refId(char[].class).eval()❺;
    int pos = intVal().eval()❻;
    int _lim1 = 0;
    while (logic(relation(intId(), intId()),
                 relation(charArrAcc(refId(char[].class), intId()), charVal()))
            .eval()❼ && _lim1++ < 1000) { pos++; }
    int _lim2 = 0;
    while (logic(relation(intId(), intId()),
                 relation(charArrAcc(refId(char[].class),
                                     arithmetic(intId(), intVal())),
                          charVal()))
            .eval()❽ && _lim2++ < 1000) { len--; } ...
    return this; }

  @Argument(0)
  public static StrBuilder _arg0() {
    StrBuilder sb1 = new StrBuilder("date");
    sb1.append((Object) 10.0);
    sb1.appendSeparator("d");
    Object[] arr = new Object[] { 1.0 };
    return sb1.append("resourceBundle", arr); } }
```

Figure 3.2: An extracted template from the program in Figure 3.1. Expressions are replaced with holes.

in Figure 3.2 represents a logical relational expression that connects two relational expressions using a logical operator (`&&`, `||`). The first relational expression connects two integer variables using one of the relational operators. The second relational expression connects a char array access expression and an integer variable. The char

```
package org.apache.commons.text;...
import jattack.annotation.*;
import jattack.csutil.Helper;
import jattack.csutil.checksum.WrappedChecksum;
import jattack.exception.UnfilledHoleException;
import static jattack.Boom.*;
public class StrBuilder implements ... { ...
  static final int CAPACITY = 32;
  char[] buffer;
  private int size;
  private String newLine, nullText;
  public StrBuilder(final String str) {
    if (nullText❶ == null) { buffer = new char[CAPACITY❷]; } ... }

  public StrBuilder trim() {
    if (size <= -1838784853❸) { return this; }
    int len = CAPACITY❹;
    final char[] buf = buffer❺;
    int pos = 809931165❻;
    int _lim1 = 0;
    while ((len > _lim1 && buf[size] != 'Z')❼ && _lim1++ < 1000) { pos++; }
    int _lim2 = 0;
    while ((pos <= _lim2 || buf[size - 1312433786] > '0')❽ && _lim2++ < 1000) {
      len--; } ...
    return this; }

  public static StrBuilder _arg0() {
    StrBuilder sb1 = new StrBuilder("date");
    sb1.append((Object) 10.0);
    sb1.appendSeparator("d");
    Object[] arr = new Object[] { 1.0 };
    return sb1.append("resourceBundle", arr); }

  public static void main(String[] args) {
    WrappedChecksum cs = new WrappedChecksum();
    StrBuilder rcvr = _arg0();
    cs.update(rcvr);
    for (int i = 0; i < 100_000; ++i) {
      try { cs.update(rcvr.trim());
      } catch (UnfilledHoleException e) { throw e;
      } catch(Throwable e) { cs.update(e.getClass().getName()); } }
    cs.update(StrBuilder.class);
    Helper.write(cs.getValue()); } }
```

Figure 3.3: A concrete program generated from the template in Figure 3.2 by filling in the holes, which crashed the OpenJ9 JIT compiler.

array access expression selects an available variable of type `char[]` as the array and utilizes an integer variable as the index value to retrieve the corresponding element from the array. The last hole (line 28) in Figure 3.2 represents a similar expression but it uses an arithmetic expression as the index of the char array access expression. The arithmetic expression applies one of the arithmetic operators (`+`, `-`, `*`, `/`, `%`) on an integer variable and an integer literal (i.e., constant).

A template must have an entry method that is the start of the execution (see Section 2.3.3), annotated with `@Entry` as shown in the template (method `trim`). One of the key challenges is to obtain an argument for the method `trim`, i.e., an instance on which the method is to be invoked, so that the template can be executed. In our example, since the entry method `trim` is an instance method, the only required input is an instance of the template class `StrBuilder` that declares the method. To provide inputs to the entry method, LeJit inserts a public static *argument method*, annotated with `@Argument` (method `_arg0`). Thus, the argument method `_arg0` instantiates a `StrBuilder` and returns the instance after invoking a sequence of methods (line 33–37) in Figure 3.2. Our key insight in this direction is to capture the sequence of methods during testing of the entry method `trim`; tests can be either existing hand-written tests or automatically-generated tests (e.g., using Randoop). The sequence of methods to return an instance of class `StrBuilder` (line 33–37) in Figure 3.2 is obtained from a generated test.

Following JAttack, LeJit generates programs by executing the template from the entry method defined in the template (see Section 2.3.3). When LeJit reaches an unfilled hole the first time, it randomly picks a valid expression within the bounded search space defined by the hole. Once LeJit has filled all reachable holes, it outputs a generated program. Figure 3.3 shows an example generated program from the template in Figure 3.2. In the figure, the hole API and the concrete expression generated to fill it share the same circled number, indicating a match between them. The generated program can be executed directly, as LeJit also generates a `main` method (line 34) in Figure 3.3, which invokes the entry method using an instance of the class that is converted into a template. This instance is returned from `_arg0` (line 36) in Figure 3.3. The `main` method repeatedly invokes the entry method in a `for` loop (line 38–41) in Figure 3.3. The large number of iterations is necessary to trigger JIT compiler optimizations in Java since the JIT compiler triggers and starts to optimize code only when a method becomes "hot", i.e., frequently executed. To encode the program behavior during execution, the `main` method hashes and saves the argument values, return values, or any thrown exceptions from each iteration, and the final class state (i.e., static fields) of the template. These hashes are used to generate a checksum, which is the final output of the execution.

To perform differential testing [87], LeJit executes every generated program using various JIT compilers and compare their outputs. The program in Figure 3.3 gave the same output using HotSpot and GraalVM, but it crashed the OpenJ9 JIT compiler. The IBM developers confirmed that the crash is due to a bug in the OpenJ9 JIT compiler and the way it handles array index out-of-bounds.

## 3.3 LeJit Framework

The LeJit framework has five key phases: (a) *collection*, (b) *extraction*, (c) *generation*, (d) *testing*, and (e) *pruning*, as illustrated in Figure 3.4. First (Section 3.3.1), LeJit collects a list of methods from the given code and obtains tests that can be used to create meaningful inputs to these methods. Then (Section 3.3.2), treating each method in the list as an entry method, LeJit selects the input that can be used to invoke the method, and extracts a template from the Java class that defines the method. Next (Section 3.3.3), LeJit executes each template with the selected inputs to the entry method to generate concrete Java programs. After that (Section 3.3.4), LeJit executes the generated programs through the entry method with the same selected inputs, using different Java JIT compilers for differential testing [87]. Finally (Section 3.3.5), LeJit prunes the detected crash and cases that lead to inconsistent outputs across various JIT compilers as to minimize false positives,

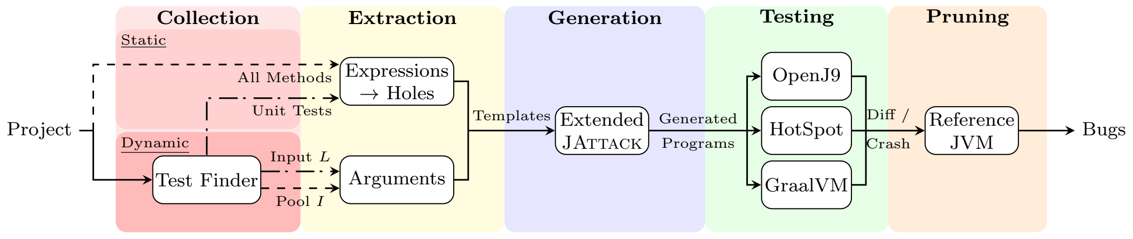

Figure 3.4: The overview of LeJit. Dotted-dashed lines: test-based approach; dashed lines: pool-based approach.

and then reports detected bugs.

### 3.3.1 Collection

In the collection phase, LeJit collects a list of methods from the given code to be used as template entry methods. LeJit then obtains tests for each of the methods, which will be used to obtain objects that can be used as arguments to the entry method.

We developed two approaches to collect a list of entry methods and to obtain code sequences that create arguments for entry methods.

**Test-based**. We use automated test generation to generate a large number of unit tests for all the classes in the given code. We utilize *the last method call* in the unit test as the entry method. As such, we can save the code sequence leading up to the method call as a way to construct arguments for that method. This approach ideally results in the same number of entry methods as the number of unit tests generated, and each entry method is associated with the saved code sequence as the input to the method.

**Pool-based**. Instead of using generated unit tests as in the previous approach, we save all prefixes of generated tests in the pool-based approach; each prefix creates an object that we add to an object pool. This object pool stores all code sequences produced during a test generation run, where each code sequence ultimately returns an instance of a class defined within the project (the object returned by the final method call in the sequence). We organize the pool using a mapping that associates each class with all the code sequences that can instantiate that class. This approach parses all the Java classes in the given project and obtains all the methods from these classes, and then uses all of the methods as entry methods.

Note that the pool-based approach creates a superset of objects created by the test-based approach, but the entry methods are different (as described above). We compare these two approaches in our evaluation to discover if they are complementary, i.e., if each leads to valuable inputs during compiler testing.

### 3.3.2 Extraction

For every entry method in the list provided by the collection phase, LeJit creates a template from the class that declares the method. Figure 3.5 shows the overall algorithm for LeJit to extract a template from a given entry method. The input to the function `Extract` is the entry method $M$, and either the collected inputs to $M$, if using the test-based approach, or the pool of inputs for all methods, if using the pool-based approach, is represented by $L$ or $I$, respectively. The output is the extracted template $\bar{t}$.

The function `Extract` starts by finding the original class $C$ that declares the entry method in the given Java code (line 6) and initializes template $\bar{t}$ as a clone of $C$ (line 7). Next, for the class, LeJit recursively converts every expression in every method (obtained from `GetAllExprs(`$\bar{t}$`)`) into a hole. Next, `Extract` replaces each expression $\bar{e}$ in $\bar{t}$ with a hole API call (i.e., Java method that represents a hole) by calling the function `Convert` (line 9) and then replacing the expression into the hole in place (line 10). Not only does LeJit convert each expression into a hole, it can also selectively create holes for some types of holes. Although we empirically evaluate impact of various types of holes, we assume in this algorithm that we convert each expression into a hole w.l.o.g.

The function `Convert` takes an expression $\bar{e}$ and its depth $\bar{d}$ as input and

```
1: Input: $M$: the entry method
2: Input: $L$: the collected input to the entry method (test-based only)
3: Input: $I$: the pool of inputs to all methods (pool-based only)
4: Output: the extracted template
5: function EXTRACT($M$, $L$, $I$)
6:    $C \leftarrow$ GETCLASSDECLARING($M$)
7:    $\bar{t} \leftarrow$ CLONE($C$)
8:    for all $e$ in GETALLEXPRS($\bar{t}$) do
9:       $e' \leftarrow$ CONVERT($e$, 0)
10:      REPLACE($e$, $e'$)
11:   for all $l$ in GETALLLOOPS($\bar{t}$) do
12:      INSERTLOOPLIMITER($l$)
13:   if $L$ then
14:      $m \leftarrow$ CREATEARGUMENTSMETHOD($L$)
15:      INSERTMETHOD($m$, $\bar{t}$)
16:   if $I$ then
17:      for all $p$ in GETALLPARAMS($M$) do                ▷ including the receiver
18:         $\tau \leftarrow$ RESOLVETYPE($p$)
19:         if ISPRIMITIVE($\tau$) then $i \leftarrow$ "<$\tau$>Val()"
20:         else if $I$.contains($\tau$) then $i \leftarrow$ RANDOMINPUTOFTYPEFROMPOOL($\tau$, $I$)
21:         else $i \leftarrow$ "null"
22:         $m \leftarrow$ CREATEARGUMENTMETHOD($p$, $i$)
23:         INSERTMETHOD($m$, $\bar{t}$)
24:   return $\bar{t}$
25: Input: $\bar{e}$: the original expression
26: Input: $\bar{d}$: the depth of $\bar{e}$
27: Output: the hole API
28: function CONVERT($\bar{e}$, $\bar{d}$)
29:   $\tau \leftarrow$ RESOLVETYPE($\bar{e}$)
30:   switch GETCATEGORYOFEXPR($\bar{e}$) do
31:      case Identifier:
32:         $h \leftarrow$ "<$\tau$>Id()"
33:      case Literal:
34:         $h \leftarrow$ "<$\tau$>Val()"
35:      case Relation:
36:         $\bar{l} \leftarrow$ CONVERT($\bar{e}$.left, $d+1$)
37:         $\bar{r} \leftarrow$ CONVERT($\bar{e}$.right, $d+1$)
38:         $h \leftarrow$ "relation(<$\bar{l}$>, <$\bar{r}$>)"
39:      ...
40:   if $\bar{d} = 0$ then return "<$h$>.eval()"
41:   else return "<$h$>"
```

Figure 3.5: Template extraction algorithm.

returns a hole API. It resolves the type of $\bar{e}$ as $\tau$, then converts $\bar{e}$ into a hole API by recursively replacing each sub-expression of $\bar{e}$ with the proper hole API call. If $\bar{e}$ is an identifier, it is converted into a `<`$\tau$`>Id` hole API call.

**Example**. The variable `str` of `String` type in the class from Figure 3.1 (line 8) is converted into `refId(String.class)` in the template from Figure 3.2 (line 10). Another `int` variable `size` (line 11) is converted into `intId()` in the same template (line 16).

If $\bar{e}$ is a literal, it is converted into a `<`$\tau$`>Val` hole API call.

**Example**. The integer number `0` in the class from Figure 3.1 (line 13) is converted into `intVal()` in the template from Figure 3.2 (line 18).

If $\bar{e}$ is not a terminal, its sub-expressions are recursively converted into hole API calls. For a relational expression $\bar{e}$, the left and right sub-expressions are converted into the hole API call $\bar{l}$ (line 36 in Figure 3.5) and $\bar{r}$ (line 37), respectively, before $\bar{l}$ and $\bar{r}$ are combined using the `relation` hole API (line 38). The operator of the relational expression is ignored because a hole API uses all available operators by default if no operator argument is provided.

**Example**. Consider the relational expression `size == 0` in the class from Figure 3.1 (line 10). The left sub-expression `size` is converted into `intId()`, and the right sub-expression is converted into `intVal()`. Then the two results are combined as `relation(intId(), intVal())` in the template from Figure 3.2 (line 15).

Other expressions that are non-terminals, e.g., arithmetic, logical, array access, etc., are converted in a similar way as a relational expression; we do not list all of them in the algorithm. Once the given expression $\bar{e}$ is converted into a hole API call, `Convert` checks if the current depth is 0. If so, it appends the `eval()` call to the hole API call and returns the resulting call as the output of the function (line 40–41). Note that `eval()` first triggers the hole API call, which returns an expression that fills the hole. Then `eval()` is called on the returned expression that evaluates to the type that the hole represents (e.g., `int`). Thus, there is only one `eval()` for the outermost hole API call.

A hole as a loop condition might introduce an infinite loop in the template class $\bar{t}$ if the hole is filled in with some expression that is always evaluated to `true` at the generation phase. Therefore, `Extract` inserts a loop limiter to restrict the maximum iterations that one loop can be executed (line 11–12 in Figure 3.5).

**Example**. Consider hole 7 in the template from Figure 3.2 (line 22), which is a loop condition. To prevent the infinite loop that may occur due to filling in random values, a loop limiter `_lim1++ < 1000` is appended to the `logic` hole to restrict the maximum iterations to one thousand times.

Once the holes are created, `Extract` then creates and inserts argument methods into $\bar{t}$, according to the selected approach in the collection phase.

**Test-based**. A public static `@Arguments` method is added to the template class $\bar{t}$ (line 14–15 in Figure 3.5). The method returns an array of all the inputs to the entry method in the order of method parameters. Consider the entry method with signature `public Quaternion multiply(final double alpha)` in the class `org.apache.commons.math4.complex.Quaternion` from open-source project math [125]. The following `@Arguments` method is generated:

```
@Arguments
public static Object[] _args() throws Throwable {
  Quaternion quaternion = new Quaternion(
      35.0, (double) 0, 57.29577951308232, -1.0);
  return new Object[] { quaternion, (double) 17 };
}
```

where `quaternion` and `17` are the inputs collected in the collection phase, i.e., extracted from a generated unit test with the entry method `multiply` as the last method call.

**Pool-based**. A public static `@Argument` method is created to provide an input for each parameter (including the receiver) of the entry method according to the type $\tau$ of the parameter (line 17–23 in Figure 3.5). If a primitive input is required, then a `<`$\tau$`>Val` hole API will be used (line 19); otherwise a reference input with the required type is randomly picked from the object pool provided from the collection phase (line 20). If the pool does not contain the type, `null` is used (line 21). Consider the entry method `trim` in the template from Figure 3.2, since it is an instance method without any parameter, only a single `@Argument` method `_arg0` (line 33–37) is created. The method `_arg0` uses a randomly picked code sequence from the object pool and returns an instance of `StrBuilder` that can be used to invoke the instance entry method `trim`.

Finally, `Extract` returns the template $\bar{t}$ (line 24 in Figure 3.5). LeJit repeats the procedure to extract a template for every entry method provided in the list from the collection phase.

### 3.3.3 Generation

LeJit obtains concrete programs from every template extracted from the previous phase. LeJit builds on JAttack to support the generation phase, i.e., generating programs through an execution-based model. Given a template $\bar{t}$, the initial global state is captured first. Then, the entry method of the template is repeatedly executed, stopping when all holes are filled or the maximum iterations $N$ has been reached. Next, the technique outputs a generated program by filling every hole with corresponding concrete code. This process repeats $M$ times to generate $M$ programs, with the template state reset after each program generation. (See Section 2.3.3 for the generation algorithm.)

**Extending template support**. LeJit enhances JAttack in several aspects to extend support for templates. (1) JAttack allows only a static method as the entry method. On the other hand, LeJit introduces the receiver object for the entry method, which allows an instance method as the entry method by passing the receiver's value from `@Argument` or `@Arguments` methods. (2) JAttack does not support non-primitive static fields in templates, as it resets a static field by saving and recovering the value. To resolve this, LeJit resets states of template classes by re-invoking static initializers (`clinit`) [12], thus allowing non-primitive static fields to be reset in templates. (3) JAttack crashes due to `UnfilledHoleException` immediately when encountering holes in static initializers, while LeJit adds extra logic to handle those exceptions when loading (including re-initializing) template classes. (4) Certain holes in constructors are not supported well by JAttack. For a `<`$\tau$`>Id` hole inside `super()` or `this()` calls from a constructor, JAttack can fill the hole with a field accessed from `uninitializedThis`, which fails bytecode verification. LeJit overcomes the limitation by tracking at which execution point in a constructor (when all `INVOKESPECIAL` and `NEW` bytecode instructions are paired up) `uninitializedThis` gets initialized and can be used. (5) LeJit introduces a number of new hole APIs for type casting and improves the checksum utility of JAttack to avoid hash collisions when hashing an object graph.

**Improving generation procedure**. LeJit makes two changes to the original generation procedure of JAttack. (1) One of the advantages of JAttack's execution-based generation over static generation is that it knows exactly what gets executed in a generated program and such information can help generate better programs. However, JAttack does not leverage the information in its implementation. It simply outputs any generated program as long as the program compiles. Instead, LeJit skips certain generated programs that are less likely to trigger JIT compiler optimizations. For instance, LeJit will skip a generated program if the execution stops even before entering the entry method due to an exception thrown from argument methods. (2) JAttack renames the class with a unique suffix in every generated program, e.g., Gen1, Gen2, etc. However, such renaming breaks circular dependencies between the template class and other classes in the same project, which makes many generated programs not compilable. LeJit disables renaming and keeps the original class name of the template for all generated programs. When executing a generated program in the testing phase, LeJit ensures that the compiled classfile of the generated program appears in the classpath prior to all the other classes of the original Java source, such that the generated program, rather than the original class with the same name in the project, will be used.

### 3.3.4 Testing

For differential testing, LeJit executes each generated program with various implementations and levels of JIT compilers, i.e., different *JIT configurations*. We define a JIT configuration as a tuple (vendor name, compiler name, version number, JVM options), for example: (Oracle, HotSpot, 20, `-XX:TieredStopAtLevel=1`) and (IBM, OpenJ9, 17.0.6, `-Xjit:optlevel=hot`). Each generated program is executed repeatedly with a large number of iterations (to trigger JIT compilation) and outputs a checksum value in the end. This checksum value is calculated by hashing the arguments provided to the entry method, the output of the return value from the entry method in each iteration, and the final state (i.e., static fields) of the entire class [144]. Then, LeJit compares the checksum values from different JIT configurations and reports a failure if it observes any difference. Additionally, LeJit reports a failure if the program crashes on some JIT configurations.

### 3.3.5 Pruning

Not every failure indicates a real issue with JIT compilers. We first discuss the failures reported due to observed inconsistent checksum values across JIT configurations. We find that most of such failures were caused by either (1) non-deterministic features of the generated program itself, e.g., random numbers, current timestamps, hashcode, etc., (2) the inconsistency between JIT configurations themselves, e.g., system property `java.vm.name` and `java.vm.info`, which contain the JVM's version information and Java options used, or (3) discrepancies between JVM implementations from different vendors, such as HotSpot and OpenJ9, which disagree on the maximum array size. To alleviate this problem, JAttack reruns the failing program twice using the interpreter mode (`-Xint`) of a single JIT configuration, while keeping the rest of the JIT configuration intact, and reports the failure only when the two reruns using interpreter mode give the exact same outputs (Section 2.3.5). However, this solution can only filter out false positive JIT compiler bugs caused by (1) but not (2) or (3). LeJit improves the filtering by (a) using various JVMs (e.g., HotSpot and OpenJ9) and (b) increasing the number of reruns of the failing program from one to three times. If any of the reruns using interpreter mode still shows inconsistent checksum values across various JVMs, LeJit considers the failure to be not related to JIT compilers and thus ignore it as a false positive.

In addition to inconsistency of checksum values across JIT configurations, crash in execution is another type of failure. When a generated program crashes while being executed using a particular JIT configuration, JAttack always labels it as a bug (Section 2.3.5). However, not all crashes are caused by issues with the JVM and therefore not all are worth reporting to developers. Some crashes are `UnfilledHoleException`, which occur due to unfilled holes in the program, which are left as API method calls during the generation phase but are reached during execution in the testing phase. In theory, such cases may be caused by incorrect JIT compilation that leads to a mismatch in program behavior between the generation and testing phases, which we want to report as a bug. However, many of these cases result from the aforementioned three reasons that cause inconsistent checksum values between JIT configurations. For example, non-deterministic features such as current timestamps may have inconsistent values between the generation phase and testing phase. This inconsistency can cause disagreement in code paths taken between the generation phase and the testing phase, e.g., when evaluating `if` conditions on timestamps, which can result in unfilled holes that were not reached during generation but were reached during testing. To address this issue, LeJit reruns the generated program with various JVMs using interpreter mode if the program reports a crash due to `UnfilledHoleException`. If the program does not crash during the rerun, then LeJit reports a bug. However, if the program still crashes during the rerun, then LeJit considers the crash as a false positive and skips reporting the failure.

While the pruning approach is simple, with manual inspection on a number of cases, we found it sufficiently useful. We also compared our pruning with original JAttack's pruning. Our pruning filtered around 96%, while JAttack filters out around 60%, out of total failures.

### 3.3.6 Implementation

We implement collection of entry methods, extraction, and pruning as standalone tools. We extend Randoop [101] to obtain objects used as arguments for non-primitive types. Finally, we extend JAttack [144] to support generation and testing phases.

## 3.4 Evaluation

We assess the value of LeJit by answering the following research questions:

**RQ1**: What are the contributions of the major components of LeJit?

**RQ2**: How effective is LeJit compared with the state-of-the-art techniques?

**RQ3**: What is the impact of holes in various Java language features on LeJit's bug detection?

**RQ4**: What is the impact of different types of templates on LeJit's bug detection?

**RQ5**: What critical bugs does LeJit detect in Java JIT compilers?

We first describe the experiment setup (Section 3.4.1) and then answer each of the research questions (sections 3.4.2-3.4.6).

### 3.4.1 Experiment Setup

**Collection**. We use open-source projects as the main input to LeJit for extracting templates. An alternative was to generate Java programs using one of the techniques

Table 3.1: Project information and number of holes per hole type. PrimitiveId contains all the `<`$\tau$`>Id` holes, and PrimitiveVal contains all the `<`$\tau$`>Val` holes, where $\tau$ is one of the primitive types in Java. # Loops is the number of loop limiters.

| **Project** | **# Holes** | | | | | | | **# Loops** | $\sum$ |
|---|---|---|---|---|---|---|---|---|---|
| | **PrimitiveId** | **PrimitiveVal** | **Array** | **Arithmetic** | **Shift** | **Relation** | **Logic** | | |
| vectorz | 1,585,341 | 498,470 | 187,333 | 468,965 | 3,115 | 207,971 | 7,960 | 112,319 | 2,959,155 |
| math | 969,150 | 391,390 | 83,093 | 275,034 | 6,392 | 114,672 | 10,547 | 67,306 | 1,850,278 |
| lang | 984,373 | 464,883 | 73,260 | 93,359 | 1,318 | 190,413 | 15,680 | 77,790 | 1,823,286 |
| text | 246,909 | 86,419 | 8,648 | 34,378 | 0 | 48,976 | 3,462 | 11,943 | 428,792 |
| compress | 178,754 | 89,706 | 9,296 | 17,012 | 2,668 | 25,710 | 3,325 | 10,163 | 326,471 |
| zxing | 93,637 | 74,610 | 10,368 | 23,204 | 1,282 | 17,740 | 3,364 | 6,937 | 224,205 |
| codec | 35,414 | 36,528 | 5,420 | 8,103 | 1,597 | 3,831 | 373 | 1,335 | 91,266 |
| statistics | 31,753 | 7,271 | 141 | 7,825 | 0 | 5,110 | 507 | 235 | 52,607 |
| jfreechart | 6,253 | 2,920 | 287 | 1,190 | 12 | 539 | 78 | 182 | 11,279 |
| numbers | 6,441 | 2,082 | 22 | 891 | 140 | 1,065 | 154 | 123 | 10,795 |
| $\sum$ | 4,138,025 | 1,654,279 | 377,868 | 929,961 | 16,524 | 616,027 | 45,450 | 288,333 | 7,778,134 |

for testing traditional Java compilers [38, 55], but open-source projects cover a much broader range of Java features. We search GitHub [54] for 1,000 Java open-source projects with the most stars, and we also include projects with at least 20 stars that belong to several popular organizations, e.g., Apache, Google, etc. In total, we collected 1,793 projects. We further filter by keeping the projects that (1) use the Maven [121] build system; (2) have a license that permits our use; and (3) have tests. After this step, there were 161 projects. Then, we attempt to build each project from its source and create a fat jar [107] for each project. We filter out any projects that cannot be packaged this way. Lastly, we exclude some projects that are not compatible with LeJit's toolchain, e.g., ASM [100], JavaParser [68], and Randoop [101]. Eventually, there are 62 projects for use.

We run LeJit once using the pool-based approach with all the 62 projects but stop LeJit early, before the generation phase, in order to collect holes from extracted templates. We next select the top ten projects with the most holes and loop limiters (used to avoid introducing infinite loops; see Section 3.3.2) in the extracted templates. Table 3.1 shows the ten open-source Java projects and associated numbers of holes and loop limiters; we show the number of holes for each *hole type*.

In the test-based approach, we configure the test generation to obtain 5,000 unit tests for each project or for 30 minutes, whichever comes earlier. In the pool-based approach, we always run test generation for 30 minutes. Lastly, we use Eclipse Temurin 11.0.18 (Adoptium OpenJDK build) in the collection phase, including building open-source projects, test generation, and running LeJit itself. We select this lower version of Java in order to maximize compatibility with open-source projects and LeJit's toolchain such as Randoop, i.e., being able to compile most projects into fat jars and to run Randoop with the projects, with a Java version.

**Generation**. We generate ten programs from each template, with a three-minute timeout. We also set a one-minute timeout in the testing phase for executing each generated program. (We change the value to 50 seconds when later comparing against JITfuzz for fair comparison.) We use Oracle JDK 17.0.6 to execute templates in the generation phase. We select a different JDK version for additional differential testing between the generation and testing phases.

**Testing**. We test a wide range of JDKs with different vendors and versions during our experiments. When evaluating LeJit alone including its variants, we use Oracle JDK 20 (HotSpot default and level 1), IBM Semeru 17.0.6.0 (OpenJ9 default and hot level), and GraalVM Enterprise Edition 22.3.1 (GraalVM default) for differential testing. When comparing LeJit with JITfuzz and JavaTailor, we also include a custom build of OpenJDK jdk-17.0.6+10 (HotSpot default and level 1) (see Section 3.4.3). We collect code coverage over the JVM code using the custom build of OpenJDK. We separately rerun generated programs to collect code coverage when evaluating LeJit alone including its variants. We collect code coverage on the fly when comparing LeJit with JITfuzz and JavaTailor. JITfuzz uses coverage, so we use the same setup for all the tools.

**Pruning**. We use reference JIT configurations to rerun three times every failing program, i.e., a generated program that either has inconsistent outputs across different JIT configurations under test or has crashed in the testing phase. We report such a failing program as a bug if the rerun using reference JIT configurations

Table 3.2: Comparison of LeJit variants. All the numbers are averages.

| | # Templates | # Programs | # Failures | # Bugs | Coverage (%) | | |
|---|---|---|---|---|---|---|---|
| | | | | | C1 | C2 | HotSpot |
| LeJit$_t$ | 10,714 | 90,916 | 66 | 3.3 | 83.8 | 79.0 | 49.5 |
| LeJit$_{\text{NoTmpl}}$ | 14,854 | 14,854 | 24 | 0.7 | 83.6 | 78.4 | 47.9 |
| LeJit$_p$ | 11,921 | 99,644 | 129 | 5.3 | 84.4 | 79.6 | 50.0 |
| LeJit$_{\text{NoPool}}$ | 10,185 | 89,977 | 56 | 4.0 | 84.0 | 78.7 | 49.2 |

does not show any difference or crash (see Section 3.3.5). We use HotSpot with `-XX:TieredStopAtLevel=0` and OpenJ9 with `-Xnojit` as reference JIT configurations when pruning failures.

**Machine**. We run all experiments on a 64-bit Ubuntu 18.04.1 desktop with an Intel(R) Core(TM) i7-8700 CPU @3.20GHz and 64GB RAM.

### 3.4.2 Contribution of Major Components

We evaluate LeJit using both test-based and pool-based approaches (Section 3.3), named LeJit$_t$ and LeJit$_p$, respectively. Additionally, we define two baselines (LeJit$_{\text{NoTmpl}}$ and LeJit$_{\text{NoPool}}$) to help us understand the benefit of using templates and creating instances for entry methods.

The variant LeJit$_{\text{NoTmpl}}$ follows the same collection phase as LeJit$_t$ that collects the last method call as the entry method from every unit test generated. However, LeJit$_{\text{NoTmpl}}$ does not extract any templates, and thus not generate any programs from templates. Instead, it directly goes to the testing phase and executes the entry method a large number of times, using just the arguments in the test. This baseline (indirectly) shows the power of automatically generated tests, obtained on a randomly selected set of projects, for discovering Java JIT compiler bugs.

We design the variant LeJit$_{\text{NoPool}}$, which extracts a template for every method in a given project. The only difference (compared to LeJit$_p$) is that LeJit$_{\text{NoPool}}$ does not collect the object pool, when extracting templates, but it rather searches for public constructors, or static methods without parameters or with only primitive parameters,

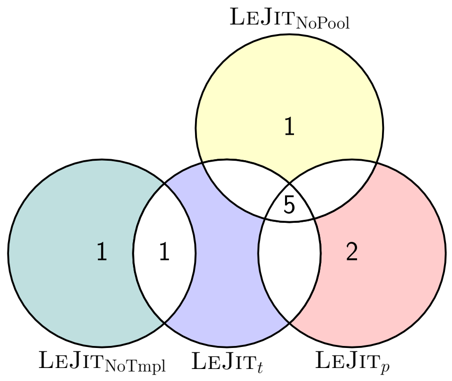

Figure 3.6: The overlap of bugs detected by LeJit variants. LeJit$_{\text{NoTmpl}}$: no templates/generated programs; LeJit$_t$: LeJit with Test-based approach; LeJit$_p$: LeJit with Pool-based approach; LeJit$_{\text{NoPool}}$: enhanced JAttack.

or `null` to construct reference arguments. LeJit$_{\text{NoPool}}$ is a superset of the original template extraction technique presented in evaluation of JAttack (Section 2.4.2); LeJit$_{\text{NoPool}}$ supports more types of holes and more entry methods than JAttack. LeJit$_{\text{NoPool}}$ shares the same generation and testing phases with LeJit$_p$.

Table 3.2 compares the numbers of generated programs, failures, and unique bugs reported. Note that all the numbers in the table represent averages from three runs. The various LeJit variants exhibit differences in their running times. Specifically, the slowest variant (LeJit$_p$) required around six days for a single run, on average. The first two rows compare LeJit$_t$ and LeJit$_{\text{NoTmpl}}$. LeJit$_t$ executes much more programs than LeJit$_{\text{NoTmpl}}$, since LeJit$_{\text{NoTmpl}}$ does not extract templates to generate programs. LeJit$_t$ also slightly increases code coverage in C1 and C2 (separate optimizing compilers within HotSpot), as well as in the entire HotSpot. However, it was interesting to observe that LeJit$_{\text{NoTmpl}}$ can even find an average of 0.7 bugs per run. As seen from the second two rows, LeJit$_p$ and LeJit$_{\text{NoPool}}$ execute a comparable number of programs and both find a few bugs. LeJit$_p$ achieves both higher coverage and higher number of bugs on average.

Figure 3.6 shows all the bugs we found from the variants and the overlap of different variants. We do not include two bugs found in our preliminary experiments and three bugs found during experimenting with various template types (Section 3.4.5). We can see that both automated generation of instances and template extraction contribute to detecting the bugs. LeJit$_t$ and LeJit$_p$ together miss two bugs that LeJit$_{\text{NoTmpl}}$ and LeJit$_{\text{NoPool}}$ find. Interestingly, LeJit$_{\text{NoTmpl}}$ without any templates or holes finds two bugs, which shows the effectiveness of traditional automated test generation even for domain that is not originally targeted.

### 3.4.3 Comparison with the State-of-the-art

We compare LeJit with JITfuzz [136] (version `3dc8f91`), a state-of-the-art technique for automated Java JIT compiler testing. Additionally, we compare LeJit with JavaTailor [153] (version `bf9421f`), a history-driven test program synthesis for testing JVM. Although JavaTailor does not target JIT compilers per se, it is worth learning about the relation and potential overlap between LeJit and JavaTailor.

**JITfuzz vs. LeJit**. To compare against JITfuzz, for a given project, we need to provide JITfuzz an *initial class* as the seed, as well as a test class as a starting point to execute the mutated programs from those seeds. Following the same methodology as in the previous work [136], we first identify ten classes in the given project with the highest cyclomatic complexity, and we pick the initial class randomly from these ten classes. Since the original work did not mention how the test class should be selected, we randomly pick a test class that imports and instantiates the initial class. To ensure a fair comparison, we run JITfuzz with the same ten projects (Section 3.4.1). During our preliminary experiment, we found that JITfuzz does not support tests using JUnit 5 [127], so we manually migrated the picked test classes to JUnit 4 [72] in five projects (other projects already used JUnit 4).

We use LeJit$_t$, i.e., with the test-based approach, in order to better control end-to-end running time by specifying the number of generated tests. (The pool-based approach uses all the available methods in the projects, which makes it hard to estimate the time needed.) We run both tools for the same length of time, around six

Table 3.3: Comparison of JITfuzz and LeJit. *The bug was already found before using JITfuzz so we did not report it again.

| | # Programs | # Failures | # Bugs | Coverage (%) | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | C1 | | C2 | | HotSpot | |
| | | | | Func. | Line | Func. | Line | Func. | Line |
| JITfuzz | 45,352 | 2,115 | *1 | 70.8 | 69.7 | 67.2 | 64.3 | 36.6 | 44.6 |
| LeJit$_t$ | 96,626 | 97 | 0 | 78.9 | 77.7 | 74.6 | 72.5 | 39.4 | 48.0 |

days, which is longer than used in the JITfuzz evaluation [136] and in recommended practice [74], while also matching the end-to-end running time of LeJit$_t$. We use the same timeout, 50 seconds, which is the default setting of JITfuzz, for executing each single generated program. JITfuzz requires custom debug builds of OpenJDK with AFL++ toolchain to work, because it needs to collect runtime coverage of JIT compiler source code [136]. Thus, we build OpenJDK jdk-17.0.6+10 from source [99] with `--enable-debug` and `--enable-native-coverage` and use the debug build as the JIT compiler under test. Note that LeJit works on both debug and release builds. We use a debug build for fair comparison (we already ran LeJit on multiple released binaries in Section 3.4.2). Also, JITfuzz does not use differential testing but detects only crashes, so we do not use OpenJ9 and GraalVM for LeJit for a fair comparison; instead we use only default level and level 1 of HotSpot from the custom debug build of OpenJDK for differential testing required by LeJit. We collect code coverage of C1 (`src/hotspot/share/c1/*`), C2 (`src/hotspot/share/opto/*`), and the entire HotSpot (`src/hotspot/*`).

Table 3.3 compares the results from both tools. Note that all the numbers in the table represent averages from three runs. JITfuzz reports 2,115 failures out of 45,352 programs that have been generated and executed. On the other hand, LeJit executes 96,626 programs and reports 97 failures. We then analyze and inspect the failures from both tools. Both tools do not detect new bugs. All the 2,115 failures reported by JITfuzz are assertion failures (which are checked on debug builds only). We group the assertion failures by stack traces and error lines in source code within

HotSpot, and there are only two unique failures. Both are duplicates of an existing bug JDK-8280126 [97] on optimizing irreducible loops. We do not find any bug from LeJit's failures. We believe the reason for this finding is that we collect code coverage on the fly for the debug build, which impacts the way JIT compilers optimize generated programs. LeJit detects a number of HotSpot bugs in other experiments we perform using non-debug builds (Section 3.4.2). LeJit increases line coverage of C1 by 8.0%, C2 by 8.2%, and HotSpot by 3.4% compared to JITfuzz.

**JavaTailor vs. LeJit**. JavaTailor [153] performs history-driven test program synthesis to test JVM implementations. More precisely, JavaTailor uses previously reported bugs as seeds to synthesize diverse test programs by combining ingredients from historical bug-revealing programs. JavaTailor was shown efficient for testing JVM implementations and here we explore if it can also discover JIT compiler bugs.

We ran JavaTailor three times in the default configuration until completion (∼8h each run). We inspected the three runs in detail and concluded that findings are similar across runs, thus no further runs were warranted. We used two versions of Java, as JavaTailor also performs differential testing: IBM Semeru 17.0.6.0 (OpenJ9 default level) and a custom build of OpenJDK jdk-17.0.6+10 (HotSpot default level) like in the previous section. We pick these two versions because HotSpot and OpenJ9 were used by JavaTailor's authors in their evaluation, and we use the custom build of OpenJDK because we need to collect code coverage of JIT compilers and compare with LeJit.

As a result of each run, JavaTailor outputs a diff log. We could not find any existing scripts for processing the diff logs, so we wrote our own to help us classify failures and perform inspection. Additionally, we wrote scripts to help us try to reproduce each of the reported failures.

JavaTailor reported 102 differences in the diff log (and each diff corresponds to one class file that is executed with two JVMs and produces different results). We semi-automatically classified the reported cases into 7 groups. Table 3.4 shows num-

Table 3.4: Number of cases of each group reported from JavaTailor.

| Group | NoRep. | NonDet. | DiffText | VerifyError | DiffException | NoException | Misc. |
|---|---|---|---|---|---|---|---|
| Number | 5 | 18 | 39 | 2 | 24 | 7 | 7 |

ber of cases of each group. NoRep. includes cases that show no differences when we tried to reproduce the difference. NonDet. includes cases that non-deterministically pass or fail (e.g., due to elapsed time being in the output) and are not revealing any bug. DiffText includes cases that are only reported with different text across JVMs, but the reported issue is actually the same. VerifyError includes cases when bytecode verification failed in both JVMs, but the messages were different. DiffException includes cases when exceptions are printed in a different order across JVMs. NoException includes cases when only one of the JVMs throws an exception, but our further inspection showed that these cases were caused by flags that have different default values across JVMs. Misc. includes single instance failures that do not fit into any other group we defined; we found one bug in this group, but the same bug was previously reported [44].

Regarding code coverage, LeJit increases code coverage of C1 by 3.3%, C2 by 4.0%, and the entire HotSpot by 0.4%, compared to JavaTailor.

In conclusion, JavaTailor can discover JVM bugs, but none were related to JIT compilers. We also found that the default reporting has many false positives. We find LeJit and JavaTailor complementary, and each could potentially benefit from the other; we discuss the combination of the two for future work in Chapter 5.

### 3.4.4 Impact of Holes in Various Language Features on LeJit's Bug Detection

In order to understand how holes in different language features contribute to bug detection of LeJit, we perform in-depth analysis on the features within extracted templates and generated programs.

We analyze three language constructs, i.e., arrays, conditional statements, and

Table 3.5: Number of templates and programs with different language features. Cond. is Conditional Statements. R.A. is Reference Arguments.

| **Project** | **# Template** | | | | | **# Generated Programs** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **Total** | **Arrays** | **Cond.** | **Loops** | **R.A.** | **Total** | **Arrays** | **Cond.** | **Loops** | **R.A.** |
| codec | 11,098 | 2,476 | 2,719 | 4,102 | 10,414 | 43,147 | 19,760 | 23,244 | 23,474 | 39,529 |
| compress | 8,269 | 1,036 | 4,133 | 1,818 | 7,794 | 62,431 | 7,585 | 35,774 | 15,713 | 58,578 |
| jfreechart | 11,041 | 215 | 2,073 | 554 | 10,656 | 61,883 | 1,894 | 13,294 | 4,117 | 59,587 |
| lang | 18,602 | 3,491 | 9,156 | 6,050 | 16,136 | 117,061 | 28,995 | 78,002 | 53,117 | 100,349 |
| math | 20,692 | 3,964 | 10,116 | 5,311 | 18,097 | 153,912 | 33,044 | 84,054 | 43,810 | 139,857 |
| numbers | 18,021 | 826 | 6,473 | 2,449 | 1,971 | 98,218 | 8,260 | 63,496 | 15,599 | 9,322 |
| statistics | 10,010 | 11 | 4,396 | 159 | 9,188 | 44,181 | 37 | 34,634 | 810 | 38,622 |
| text | 11,532 | 2,423 | 5,105 | 3,506 | 10,752 | 66,395 | 18,975 | 46,965 | 31,068 | 61,473 |
| vectorz | 23,005 | 6,031 | 11,277 | 6,904 | 21,718 | 175,814 | 49,594 | 94,726 | 58,759 | 165,803 |
| zxing | 10,925 | 1,660 | 3,767 | 2,375 | 10,145 | 63,136 | 13,440 | 30,993 | 19,417 | 57,703 |
| $\sum$ | 143,195 | 22,133 | 59,215 | 33,228 | 116,871 | 886,178 | 181,584 | 505,182 | 265,884 | 730,823 |

loops, and one other language feature, i.e., reference arguments. If a filled hole is inside a language construct (e.g., a hole is inside a loop), then we say the generated program that contains the filled hole *has* the language construct, and we also say the associated template from which the generated program is generated *has* the language construct. Similarly, we also measure how many templates and generated programs use an entry method that needs an argument of non-primitive (reference) type, which means the arguments are obtained by generated tests. We say such templates and generated programs have reference arguments. Table 3.5 shows the numbers of templates and generated programs that use the four language features. We can see that a substantial number of templates and programs need non-primitive arguments.

Similarly, we say a bug *has* a language feature if any generated program that exposes the bug (i.e., failure due to bug) has the language feature. Note that we do not claim that the presence of a feature implies that the bug is related to the feature or the feature is the root cause of the bug. Table 3.6 shows the numbers of failures due to bugs and unique bugs that use various language features. In conclusion, LeJit well explores the four language features and holes in each of these features contribute to the unique bugs discovered.

Table 3.6: Impact of Java language features on bugs. Cond. is Conditional Statements. R.A. is Reference Arguments. *Bugs may repeat across projects, and we show the number of unique bugs across all projects.

| **Project** | **# Failures due to Bugs** | | | | | **# Bugs (Unique)** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **Total** | **Arrays** | **Cond.** | **Loops** | **R.A.** | **Total** | **Arrays** | **Cond.** | **Loops** | **R.A.** |
| codec | 5 | 1 | 3 | 1 | 5 | 3 | 1 | 2 | 1 | 3 |
| compress | 6 | 1 | 5 | 1 | 6 | 2 | 1 | 1 | 1 | 2 |
| jfreechart | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| lang | 24 | 6 | 18 | 14 | 23 | 2 | 1 | 2 | 2 | 2 |
| math | 21 | 11 | 19 | 10 | 13 | 6 | 4 | 6 | 4 | 4 |
| numbers | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| statistics | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| text | 2 | 1 | 2 | 2 | 2 | 2 | 1 | 2 | 2 | 2 |
| vectorz | 107 | 42 | 95 | 91 | 93 | 5 | 3 | 5 | 5 | 5 |
| zxing | 8 | 8 | 2 | 8 | 0 | 1 | 1 | 1 | 1 | 0 |
| $\sum$ | 174 | 70 | 144 | 127 | 142 | 10* | 7* | 8* | 8* | 9* |

Table 3.7: Results when using a specific set of types of holes.

| | `<`$\tau$`>Id()` | `<`$\tau$`>Val()` | `arithmetic()` & `shift()` | `relation()` & `logic()` | **All** |
|---|---|---|---|---|---|
| #Templates | 13,090 | 12,139 | 13,606 | 13,682 | 12,061 |
| #Programs | 105,759 | 87,230 | 59,917 | 64,906 | 102,670 |
| #Failures | 106 | 29 | 39 | 70 | 131 |
| #Bugs | 3 | 1 | 3 | 4 | 4 |

### 3.4.5 Impact of Template Types on LeJit's Bug Detection

To explore how different types of templates affect LeJit's bug detection, we extract different sets of templates from the same ten projects (Section 3.4.1). We modified the template extraction algorithm (Figure 3.5) so that each extracted set of templates contains a single set of specific types of holes out of (1) `<`$\tau$`>Id`, (2) `<`$\tau$`>Val`, (3) `arithmetic()` and `shift()`, (4) `<`$\tau$`>relation` and `<`$\tau$`>logic`. We also extract a set of templates with all types of holes, which is the default setting. Other than the extraction phase, we use the same methodology and configuration as described in Section 3.4.1 to run LeJit with the five sets of templates. Table 3.7 shows the numbers of templates, generated programs, reported failures and bugs from all five sets of templates. The set of templates with all types of holes reports the most number of bugs. Out of the other four sets of templates with only a single set of holes, the `relation()` and `logic()` holes reports the most number of bugs, but even

Table 3.8: Detected bugs in HotSpot, OpenJ9 and GraalVM using LeJit; 11 bugs were previously unknown.

| JVM | Bug ID | Type | JDK Versions | Status | CVE | Duplicates |
|---|---|---|---|---|---|---|
| GraalVM | GR-45498 | Diff | 17, 20 | Fixed | - | - |
| HotSpot | JDK-8301663 | Diff | 18, 19, 19.0.2 | Fixed | - | JDK-8288064 |
| | JDK-8303946 | Diff | 8, 11, 17, 19, 20, 21 | Confirmed | - | - |
| | JDK-8304336 | Diff | 17, 19, 20, 21 | Fixed | CVE-2023-22044 | - |
| | JDK-8305946 | Crash | 17, 19, 20, 21 | Fixed | CVE-2023-22045 | - |
| | JDK-8325216 | Crash | 17, 18, 19, 20, 21 | Fixed | - | JDK-8319793 |
| OpenJ9 | 17066 | Crash | 8, 11, 17, 18 | Fixed | - | - |
| | 17129 | Diff | 8, 11, 17, 18 | Fixed | - | - |
| | 17139 | Diff | 8, 11, 17, 18 | Fixed | - | - |
| | 17171 | Crash | 11, 17, 18 | Fixed | - | - |
| | 17212 | Crash | 8, 11, 17, 18 | Fixed | - | 15363 |
| | 17249 | Diff | 8, 11, 17, 18 | Fixed | - | - |
| | 17250 | Diff | 17, 18 | Fixed | - | - |
| | 18802 | Crash | 8, 11, 17, 21 | Fixed | - | 17045 |
| | 18803 | Crash | 11, 17, 21 | Fixed | - | - |

the simplest set of holes, i.e., constant replacement (`<`$\tau$`>Val`), plays an important role. Furthermore, we discovered three additional JIT compiler bugs using these various sets of templates.

### 3.4.6 Detected Bugs

Table 3.8 lists the bugs that LeJit detected. So far, we have discovered and reported 15 bugs, 11 of which are previously unknown, including two CVEs. We show (in Figure 3.7–3.10) and describe four bugs that encompass a variety of JIT compiler issues.

**Arithmetic mis-compilation**. A mis-compilation occurred when the OpenJ9 JIT compiler performed a modular operation with parameter passing (Figure 3.7). We discovered the bug using a template created from math [125]. The issue lies in an incorrect reuse of a register whose value changes after a floating-point remainder operation.

**Incorrect elimination of range checks**. From a template extracted from math [124], we discovered a HotSpot JIT compiler mis-compilation bug where the range check for array accesses was incorrectly eliminated, which missed throwing exceptions and

```java
public class C {
  double q0, q1, q2, q3;
  C(double a0, double a1, double a2, double a3) {
    q0 = a3; q1 = a1; q2 = 0; q3 = 0; }
  static double m(double d) { C c = new C(0, 1.0, 0, d % d); return c.q1; }
  public static void main(String[] args) {
    double sum = 0;
    for (int i = 0; i < 100_000; ++i) {
      sum += m(1.0); } // m(1.0) expected to be 1.0 returns 0.0.
    System.out.println(sum); } } // expected 100000.0.
```

Figure 3.7: Arithmetic mis-compilation (OpenJ9 17129).

```java
public class C {
  static void m(int n) {
    int[] a = new int[n];
    for (int i = 0; i < 1; i++) { int x = a[i % -1]; } }
  public static void main(String[] args) {
    int count = 0;
    for (int i = 0; i < 1000; ++i) {
      try { m(0);
      } catch (ArrayIndexOutOfBoundsException e) { count += 1; } }
    System.out.println(count); } } // expect 1000.
```

Figure 3.8: Incorrect elimination of range checks (HotSpot JDK-8304336).

produced incorrect results (Figure 3.8). Upon reporting the bug, Oracle developers promptly confirmed the issue. They classified the bug as a CVE and rolled out the fix in the next Critical Patch Update.

**Erroneous loop condition evaluation**. Execution of OpenJ9 JIT-compiled code faced a situation where a loop condition was incorrectly evaluated as true, enabling the loop body to run (Figure 3.9). However, the loop body should never execute, and this correct behavior was observed in non-JIT executions. LeJit flagged this issue as a bug using a template from codec [122]. IBM developers confirmed the bug within a day.

**Standard library mis-compilation**. An incorrect output occurred when using GraalVM JIT compilation with the String `getBytes` method (Figure 3.10). The

```
public class C {
  static int m(int len) {
    int[] arr = new int[8];
    for (int i = 10000000, j = 0; (boolean) (i >= 1) && j < 100; i--, j++) {
      // should not enter inner loop.
      for (int k = 0; len < arr.length; ++k) { int x = 1 / 0; }
    } return 0; }
  public static void main(String[] args) {
    int sum = 0;
    for (int i = 0; i < 100_000; ++i) {
      try { m(13);
      } catch (ArithmeticException e) { sum += 1; } }
    System.out.println(sum); } } // expected 0.
```

Figure 3.9: Erroneous loop condition evaluation (OpenJ9 17249).

```
static import java.nio.charset.StandardCharsets;
public class C {
  static int m(String s) {
    byte[] arr = s.getBytes(ISO_8859_1);
    return arr[2]; }
  public static void main(String[] args) {
    long sum = 0;
    for (int i = 0; i < 10_000_000; ++i) { sum += m("\u8020\000\000\020"); }
    System.out.println(sum); } } // expected 0
```

Figure 3.10: Standard library mis-compilation (GraalVM GR-45498).

generated program by LeJit emerged from a template based off code from codec [123]. The developers confirmed the bug within one day.

Bugs detected with LeJit are presented in an easily digestible manner. Generated programs are easy to minimize and understand, because LeJit extracts templates from real-world Java programs and the minimum example programs we submitted are Java source code. Developers were able to quickly understand our reports and reproduce or further minimize source code as needed. Many reports were confirmed by the first 48 hours. In contrast, bytecode files generated by some tools require substantial effort to understand by compiler developers [98].

## 3.5 Limitations

When we compared LeJit with JITfuzz, we used only the test-based approach, and we were unable to successfully run another tool: JOpFuzzer [69]. Also, when we evaluated LeJit variants, we did not attempt to match end-to-end duration of LeJit$_{\mathrm{NoTmpl}}$, or LeJit$_{\mathrm{NoPool}}$, and end-to-end duration of LeJit$_t$, or LeJit$_p$. The randomness could impact the experiment results, so we run each experiment three times, as we already described.

LeJit has enhanced JAttack to support non-primitive static fields in templates by re-initializing the template class, but LeJit is limited on re-initializing other classes in dependencies. While JAttack's exception handling has been enhanced to handle exceptions thrown from class loading and initializing, LeJit does not handle errors directly thrown from JVM in the generation and testing phase, e.g., `StackOverflowError`, `OutOfMemoryError`, etc. These shortcomings sometimes left unfilled holes that were supposed to be filled, leading to false positives in the testing phase. We will further explore re-initializing all classes in dependencies and better handling of JVM errors.

**Ethical considerations**. To avoid "spamming" open-source community, we submit a bug report only when we can reproduce the bug on the latest release of the affected JDKs. We also tried our best to detect duplicates and minimize the programs that reproduce the bugs.

## 3.6 Conclusion

We presented a framework, LeJit, which enables fully automated end-to-end template-based testing of JIT compilers. LeJit can create a template from any Java method, and it automatically inserts holes and generates necessary arguments for the template. To obtain instances of complex types needed for extracted templates, LeJit uses novel techniques built on automated test generation. We have extensively evaluated LeJit by generating 90,916 programs and discovered 15 bugs in three popular and widely used compilers. Our findings show the power of automating template extraction via LeJit and the power of scaling experiments without humans in the loop, as well as complementary power compared to the state-of-the-art tools for JIT compiler and JVM testing techniques. We believe that LeJit should become an integral part of continuous testing for any Java JIT compiler.

# Chapter 4: Related Work

In this chapter, we describe the work most related to this dissertation. First, we review compiler testing in general (Section 4.1). Next, we describe work on Java JIT compiler or JVM testing using grammar-based, mutation-based and template-based approaches (Section 4.2), and also review work on testing JIT compilers of languages other than Java (Section 4.3). Then, we discuss the test oracle problem in compiler testing (Section 4.4) and test input generation in general (Section 4.5). Finally, we summarize the related work on JIT compiler verification (Section 4.6) and security on JIT compilers (Section 4.7).

## 4.1 Compiler Testing in General

There is a large body of work on compiler testing, systematically reviewed in recent surveys [25, 120]. There are mainly three approaches to generating programs as test inputs: grammar-based, mutation-based, and template-based.

The grammar-based generation [3, 9, 24, 45, 84, 85, 88, 90, 92, 106, 114, 139, 143] uses the grammar production rules of the language to generate programs. For example, Csmith [139] is a well-known tool for testing C compilers by randomly generating C programs, which found bugs in mainstream compilers [140, 141]. Mutation-based fuzzing [22, 31–33, 46, 52, 63, 69, 76, 105, 136, 153] creates new programs to test compilers by mutating existing programs. LangFuzz [63] is a representative early work and implemented to test JavaScript engines. It randomly picks code fragments in existing programs and replaces the fragment with other possible fragments collected from existing programs. Template-based approach [34, 51, 115, 116, 152] creates programs via filling placeholders in predefined templates. For instance, SPE [152] statically exhaustively enumerates possible variable occurrences to fill the placeholders in the given skeleton program (template). JAttack is the first dynamic template-based approach to testing Java JIT compilers, and LeJit sits in between the mutation-based and template-based approaches.

## 4.2 Java JIT Compiler / JVM Testing

Existing works on testing Java JIT compilers or JVM mainly fall in grammar-based and mutation-based approaches.

### 4.2.1 Grammar-Based

These tools use the grammar production rules of the Java language, or the Java bytecode language, to generate programs in Java source code, or in Java bytecode, respectively, as test inputs to compilers. Java* Fuzzer [9] is a grammar-based tool that generates random Java programs to detect crashes, hangs and incorrect calculations of JVM. Sirer and Bershad [114] proposed lava to generate from production grammars Java bytecode to test JVM. Yoshikawa et al. [143] proposed an approach for generating Java bytecode following grammar for testing Java JIT compilers.

### 4.2.2 Mutation-Based

This approach generates new programs by mutating existing programs, either in Java source code or bytecode. Classfuzz [32] and its follow-up Classming [33] mutate seed classfiles (in Java bytecode) by modifying their syntactic structures, e.g., renaming fields, changing modifiers, etc., and by modifying control and data flow, e.g., inserting or deleting `goto` and `return` statements. SJFuzz [137] improves Classming [33] by introducing edit distance [79] to diversify mutant programs when scheduling seeds and mutation operators. While such tools guide the mutation process using a coverage-guided algorithm, in order to maximize the coverage of the tested JVM, many crashes are still found in the early verification stage without testing deeper logic within JVM such as JIT compilers. JITfuzz [136] is also a coverage-guided Java JIT compiler fuzzer, but it introduces novel mutation operators designed for JIT compilers, e.g., scalar-replacement-activating mutator.

On the other hand, JavaTailor [153] generates programs from bug-revealing programs in previous bug reports from OpenJDK and OpenJ9. More precisely, JavaTailor combines code ingredients from these bug-revealing programs to create diverse programs in Java bytecode. VECT [52] improves JavaTailor [153] via grouping code ingredients by code vectorization and then selecting representatives of each group, so as to better diversify the generated programs. ComFuzz [142] also learns from bug-revealing programs from historical bug cases. It starts with deep learning models to generate programs and then leverages predefined mutation rules to create final programs.

Some techniques introduce new dimensions on the test oracle (which we will discuss more in Section 4.4) over the traditional mutation-based fuzzing approach. JOpFuzzer [69] uses different JIT compiler optimization options when running a mutation-created program using Java JIT compilers. It picks options according to what is mutated in the program. Artemis [80] compares various JIT compilation choices during differential testing, i.e., diverse combinations of optimized/deoptimized methods for the same sequence of method invocations.

### 4.2.3 Template-Based

To the best of our knowledge, JAttack opened a new area: template-based testing for Java JIT compilers. JAttack generates Java programs by utilizing the structures of a given template, and explores possibilities by filling holes. On the one hand, unlike the grammar-based approach that generates programs from scratch following the grammar, compiler developers can use their best intuition and knowledge when writing the template for JAttack, and have full control of the space that should be tested and the way generated programs should evolve. On the other hand, in contrast to the mutation-based approach, JAttack has several differences. (1) Mutation-based approaches use a predefined set of mutation operators. However, each hole in a template has its own set of values (and the set can be dynamically determined; see the next point). (2) JAttack executes templates to fill holes dynamically (rather than statically), which brings unique advantages: (a) uses meaningful variables to fill holes; (b) allows hole construction with runtime information, e.g., lengths of arrays; (c) allows JAttack to establish which holes are in dead code and, in the generation phase, focus on exploring the reachable holes. (3) JAttack can fill multiple holes simultaneously during the execution of the template, which is more similar to higher-order mutation [70]. Compared with existing work on template-based compiler testing in other languages, e.g., SPE [152] for C language, JAttack generates concrete programs by executing templates, instead of statically filling holes, which allows developers to use values available at runtime to construct holes. JAttack also allows richer expressions, e.g., arithmetic expressions, relation expressions, logic expressions, etc., to be generated in holes, besides variables.

LeJit complements JAttack by automatically creating templates from real-world Java programs, through converting concrete expressions into holes and generating glue code to create arguments with non-primitive types. Thus, LeJit with JAttack together blends the mutation-based and template-based approach. Although the blending is similar in nature to concepts in mutation testing [40], via creating templates and then using them to generate concrete programs, the aforementioned dynamic execution of templates by JAttack, and extracting templates and obtaining objects necessary for running templates provided by LeJit make it a novel approach that complements existing techniques.

## 4.3 JIT Compiler Testing for non-Java Languages

In addition to Java, there has been research on testing JIT compilers of other languages. Notably, a lot of research efforts have been dedicated to testing JavaScript engines. jsfunfuzz [89] is a grammar-based JavaScript engine testing tool, similar to Java* Fuzzer [9], and it follows the JavaScript grammar and generates JavaScript programs. Fuzzilli [58], a mutation-based JavaScript engine testing tool, proposes a new intermediate representation language named FuzzIL to express the desired mutation of abstract syntax trees of JavaScript programs. DIE [105] performs type-preserving and structure-preserving mutation on existing JavaScript programs that revealed bugs. Similar to our work, more recent works considered JIT compilers inside JavaScript engines. JITPicker [14] injects state probes which computes hash for variables during execution, to compare the runtime behavior between JavaScript interpreters and JIT compilers. However, it needs to modify JavaScript engines to make sure the probes will not affect JIT compiler optimization. FuzzJIT [133] mutates code elements related to JIT compilers, e.g., arrays, objects, etc., and it also generates JavaScript programs that is test-oracle integrated, i.e., the comparison between the interpreter and the JIT compiler is done inside the program.

As for other languages, Fuzzlyn [91, 92] is a grammar-based testing tool of C# JIT compilers, similar to Java* Fuzzer [9] and jsfunfuzz [89]. Ranger [106] treats the Smalltalk JIT compiler as a white-box and uses concolic testing [56, 111] on the interpreter to generate test inputs to the Smalltalk JIT compiler.

## 4.4 Test Oracles for Compiler Testing

The test oracle problem is a fundamental problem in the area of compiler testing, i.e., how to determine if the compiler compiles the given program correctly. A crash during compilation is obviously a red flag, but it is not trivial to determine correctness when the compilation finishes normally. To address the challenge, researchers have mainly used two approaches: differential testing [87] and metamorphic testing [29].

McKeeman [87] first proposed differential testing as to complement regression testing. Most of aforementioned work on compiler testing (Section 4.1–4.3), e.g., [32, 139, 152], uses differential testing to determine a mis-compilation bug when compilation does not crash. In general, it is required that at least two compilers are implemented based on the same specification, and then a program is run using these compilers separately. If the compilers do not agree on the results, then there is a bug in some compiler(s). In practice, in terms of different implementation of the same compiler specification, most researchers consider different vendors, versions, and optimization levels (including comparing interpreters and JIT compilers) [133, 136, 153]. Some researchers also incorporate domain-specific factors into differential testing when testing Java JIT compilers. JOpFuzzer [69] uses JIT compiler optimization options as a new dimension, and Artemis [80] compares various combinations of optimized/de-optimized methods for the same sequence of method invocations. Additionally, SpecTest [110] includes executable semantics, e.g., K semantics for Java [15], as an implementation of the compiler in differential testing. JEST [104] generates assertions from JavaScript language specifications in programs that will be used as test inputs to JavaScript engines, and in this way includes the language specification in differential testing.

Metamorphic testing [29] is another approach to addressing the test oracle problem, which constructs metamorphic relations that describe how a particular change to the test input impacts the test output. The most widely used relation in the area of compiler testing is the equivalence relation. Le et al. [76] proposes a method named Equivalence Modulo Inputs (EMI) to test compilers, which converts an original program into multiple equivalent, but seemingly different, programs and compares results of running these programs after them being compiled. If the results are different, then a compiler bug is found. This method allows to test even a single compiler since it does not need to compare between different compilers. Researchers have extended the concept of EMI to test C/C++ compilers [77, 90, 118], OpenCL compilers [83], OpenGL [41], Simulink compilers [35], etc. The ways to obtain equivalent variants in the aforementioned works include inserting/deleting dead code, applying predefined transformation rules, etc. Chen et al. [23] applied EMI on generated programs by Csmith [139], and used the created equivalent variants to test GCC and LLVM separately. They compared the effectiveness of discovering bugs in this way (using metamorphic testing) with the original Csmith (using differential testing), and showed that differential testing and metamorphic testing techniques can complement each other to a certain degree.

## 4.5 Test Input Generation

Compiler testing techniques generate programs as test inputs to a specific kind of software, i.e., compilers. A lot of research has also explored test input generation for general software [4, 55, 56, 101, 112]. We describe some notable works in test input generation for Java programs.

JUnit tests can be automatically generated using Randoop [101] and EvoSuite [48] for a given set of classes under test. In theory, the tests generated from such tools can be directly used to test Java JIT, and we implemented such a prototype using Randoop and evaluated its bug-discovery effectiveness within our evaluation of LeJit (see Section 3.4.2).

Popularized by QuickCheck [36], another approach is to allow developers to write generators from which valid test inputs can be obtained. ASTGen [38], UDITA [55], and Tempo [7] use the generators to exhaustively enumerate all possible paths through the generators up to a given bound. JQF [102] integrates coverage-guided fuzzing into the QuickCheck-style framework for testing Java programs. Theoretically, these tools can be used to test Java JIT compilers as well, but they would require developers to write generators for Java programs.

Additionally, symbolic execution [73] has been used in test input generation, such as DART [56], JPF [132], etc. Although such tools can precisely explore different paths and produce test inputs exercising new behaviors, the path explosion problem prevents them from being scalable.

## 4.6 JIT Compiler Verification

Aside from testing, verification is another technique to ensure correctness of software. There has been verified static compilers such as CompCert [78], CakeML [75], etc. Shingarov [113] experimented with a proof of concept formal verification of JIT compilers by symbolic execution. Guo and Palsberg [59] discussed the soundness of trace-based JIT compilers. Flückiger et al. [47] and Barrière et al. [10] formally verified speculative optimization and deoptimization in a JIT compiler. Barrière et al. [11] presented a JIT compiler model with dynamic generation of native code, which is formally verified in Coq reusing CompCert and its correctness proofs. In addition, there are other works on verifying range analysis routines of JavaScript JIT compilers with SMT solvers [18], validating semantic equivalence of assembly language output from various versions of C# JIT compilers [62], and building formally verified in-kernel JIT compilers [130, 134]. Orthogonal to these efforts on verification, our work focuses on testing, designed to check correctness of widely used Java JIT compilers, including HotSpot, OpenJ9, etc.

## 4.7 Security on JIT Compilers

JIT compilation brings unique challenges to security. Gawlik and Holz [53] reviewed JIT spraying attacks, and Lian et al. [81, 82] extended JIT spraying attack to ARM. Chen et al. [27] proposed a defense of JIT spraying attacks by controlling the execution of the JIT-compiled code, while others [28, 65, 135, 138] use randomization and obfuscation to defend from these attacks. Athanasakis et al. [6] showed that JavaScript JIT engines are exploitable using solely dynamically generated gadgets. Song et al. [117] demonstrated the feasibility of exploiting race conditions to maliciously modify code cache, and proposed secure dynamic code generation. Frassetto et al. [49] proposed a generic data-only attack against JIT compilers that enables arbitrary code-execution, and also proposed a defense to mitigate such attacks. Brennan et al. [16] demonstrated how JIT compilation can be exploited for timing side-channel attacks. Brennan et al. [17] also presented a technique for automatically detecting such JIT-induced timing side-channel attacks, and Qin et al. [108] proposed an approach to eliminating such JIT-induced timing side-channel leaks. Ansel et al. [5] introduced general mechanisms for safely and effectively sandboxing dynamic language runtimes, including JIT compilers. Niu and Tan [93] and Zhang et al. [151] proposed approaches to securing JIT compilers through control-flow integrity [1]. Although JAttack and LeJit are designed to detect any correctness bugs, we did discover several security vulnerabilities (e.g., CVE-2020-14792 [128]) inside Java JIT compilers.

# Chapter 5: Future Work

In this chapter, we describe ways in which JAttack and LeJit can be improved and extended from the five aspects, i.e., improving program generation, optimizing testing procedure, refining testing results, and testing other software systems.

## 5.1 Improving Program Generation

**Holes for method or constructor invocation**. Although JAttack has provided APIs for writing holes for many Java language features, e.g., primitive values, variables, logical expressions, etc. There are several Java language features to be supported as written using holes in JAttack, including holes for selecting methods or constructors. Such a hole can allow JAttack to both explore a wider search space of a template program and to test more JIT compiler optimizations [145, 147], e.g., method inlining.

**Smarter searching while filling holes**. Currently JAttack fills a hole by randomly choosing a value within the defined search space of the hole, and filling of multiple holes is independent to each other (although JAttack does skip filling unreachable holes due to its execution-based nature). A smarter strategy of filling holes can more efficiently generate programs that expose bugs and can also explore greater search space defined by the template. Similar to some mutation-based compiler testing techniques [136, 137], some metrics can be leveraged to guide hole filling: e.g., code coverage of compilers under test, the types/shapes of intermediate representation nodes in compilation, etc. Also, considering relations between holes, i.e., filling a hole according to how surrounding holes are filled, can maximize the effectiveness of the entire template.

**Using historical bugs cases to create templates**. LeJit creates templates from existing Java programs in open-source projects. Existing works [52, 142, 153] have shown effectiveness of discovering new bugs from using the programs revealing historical bugs. We expect the regression tests related to historical JIT compiler bugs from OpenJDK and OpenJ9 repositories to be another valuable source of creating templates.

## 5.2 Optimizing Testing Procedure

**De-duplication of templates and generated programs**. We have found multiple templates and/or generated programs may reveal the same JIT compiler bug. Therefore, de-duplicating templates and generated programs may significantly speed up the testing procedure, explore a greater search space, and potentially lead to more new bugs. On the other hand, EMI techniques [76] (see section 4.4) have shown that even semantically equivalent programs may have different impacts on compilers. JIT compilers may still optimize two semantically equivalent programs differently based on their code structure. Determining which programs are duplicated w.r.t. how JIT compilers should optimize them is worth exploring further.

**Test prioritization and test-suite reduction**. The main running time of end-to-end Java JIT compiler testing using JAttack and LeJit is spent on executing the generated tests with the tested compilers. Existing works have used techniques of test prioritization and test-suite reduction to accelerate the testing procedure [21, 26]. Similar techniques can also be used to speed up the execution of programs generated from templates.

## 5.3 Refining Testing Results

**Decreasing the false positive rate**. While we reported a failure as a potential JIT compiler bug only when rerunning at the interpreter does not show inconsistency, this rerunning is not sufficient to filter out all false positives. In our manual inspection of the reported failures after the rerunning, we still found many false positives due to non-deterministic features used in the programs, e.g., `Object.hashCode()`, randomness, current timestamps, etc. One way to to reduce the false positive rate is to instrument associated Java libraries to fix the non-deterministic values [60].

**Identifying duplicated bugs**. In our experiments, the number of unique bugs was far less than the number of failures, because multiple failures exposed a same bug. It is also time-consuming to manually check if every failure is a duplicated bug to one of the bugs we have found. Such process of identifying duplicated bugs can be partially automated via comparing their crash stack traces, or program similarity [30, 64].

**Reducing buggy programs**. Since we used real-world Java programs to create templates, the templates and generated programs were usually large and complicated. Once a generated program detects a JIT compiler bug, we always manually reduced the program before we reported the bug, so that developers could locate and fix the bug more easily. Researchers have developed several techniques for reducing and debugging programs, e.g., ddmin [149, 150], C-Reduce [109], Perses [119], and VeDebug [20], which can be applied to automate the process of reducing buggy programs.

**Flakiness of JIT compiler bugs**. Not all JIT compiler bugs we detected are reproducible each time when run due to the non-deterministic nature within JIT compilers (which is different from non-deterministic features within programs, discussed in the first paragraph of this section). For example, in one of the generated programs for template M4 (Section 2.5), we could not always observe failure (crashing the JVM) every time we run on the same JIT compiler. We could have missed this bug because of an unfortunate non-crash run. Thus, to explore and control the non-deterministic nature within JIT compilers can potentially reveal more new JIT compiler bugs (from false negatives).

## 5.4 Testing Other Software Systems

While we use Java JIT compilers to demonstrate the usefulness of template-based testing, it will be easy to migrate JAttack and LeJit to test other software systems that also take Java programs as input, such as refactoring tools (e.g., Eclipse [43]). Also, it is valuable to explore the possibilities to use template-based techniques to test JIT compilers of non-Java languages, such as C#, JavaScript, etc.

# Chapter 6: Conclusion

Compilers are an integral part of the software development toolchain, and thus their correctness is of utmost importance. However, existing techniques for testing compilers either are time-consuming or lack the flexibility for compiler developers to apply their domain knowledge effectively. To address these challenges, this dissertation introduces JAttack and LeJit: JAttack enables template-based compiler testing that allows compiler developers to write templates based on their insights, and LeJit is a unified automated testing framework around JAttack, which streamlines creation of templates from existing Java code and testing of Java JIT compilers.

JAttack allows developers to write templates in the same language as the compiler they are testing (Java), enabling them to leverage their domain knowledge to set up a code structure likely to lead to compiler optimizations while leaving holes representing expressions they want explored. JAttack executes templates, exploring possible expressions for holes and filling them in, generating programs to later be run on compilers. Through application of JAttack, we found seven bugs in the HotSpot JIT compiler. Five of them were previously unknown, including two unknown CVEs.

LeJit is an overarching automated testing framework wrapping JAttack. LeJit can create a template from any Java method, and it automatically inserts holes and generates necessary arguments for the template. To obtain instances of complex types needed for extracted templates, LeJit uses novel techniques built on automated test generation. Using LeJit, we discovered 15 additional bugs in three popular and widely used Java JIT compilers (HotSpot, OpenJ9 and GraalVM), 11 of which were previously unknown, including two unknown CVEs.

Our findings show the power of combining developers' domain knowledge (via templates) with automated testing, and we believe that JAttack and LeJit should be in integral part of the testing process for any Java JIT compiler.

# References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Conference on Computer and Communications Security*, pages 340–353. ACM, 2005. `https://doi.org/10.1145/1102120.1102165`.

[2] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007. ISBN 9780321547989.

[3] Mohammad Amin Alipour, Alex Groce, Rahul Gopinath, and Arpit Christi. Generating focused random tests using directed swarm testing. In *International Symposium on Software Testing and Analysis*, pages 70–81. ACM, 2016. `https://doi.org/10.1145/2931037.2931056`.

[4] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil Mcminn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013. `https://doi.org/10.1016/j.jss.2013.02.061`.

[5] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. Language-independent sandboxing of Just-in-Time compilation and self-modifying code. In *Programming Language Design and Implementation*, pages 355–366. ACM, 2011. `https://doi.org/10.1145/1993316.1993540`.

[6] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. The devil is in the constants: Bypassing defenses in browser JIT engines. In *The Symposium on Network and Distributed System Security*, pages 8–11. The Internet Society, 2015. `https://doi.org/10.14722/ndss.2015.23209`.

[7] Nader Al Awar, Kush Jain, Christopher J. Rossbach, and Milos Gligoric. Programming and execution models for parallel bounded exhaustive testing. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 166:1–166:28. ACM, 2021. `https://doi.org/10.1145/3485543`.

[8] John Aycock. A brief history of Just-in-Time. *ACM Computing Surveys*, 35 (2):97–113, 2003. `https://doi.org/10.1145/857076.857077`.

[9] Azul Systems, Inc. Azulsystems/JavaFuzzer: Java* Fuzzer for Android*, 2018. `https://github.com/AzulSystems/JavaFuzzer`.

[10] Aurèle Barrière, Sandrine Blazy, Olivier Flückiger, David Pichardie, and Jan Vitek. Formally verified speculation and deoptimization in a JIT compiler. In *Symposium on Principles of Programming Languages*, pages 46:1–46:26. ACM, 2021. `https://doi.org/10.1145/3434327`.

[11] Aurèle Barrière, Sandrine Blazy, and David Pichardie. Formally verified native code generation in an effectful JIT: Turning the compcert backend into a formally verified JIT compiler. In *Symposium on Principles of Programming Languages*, pages 9:1–9:29. ACM, 2023. `https://doi.org/10.1145/3571202`.

[12] Jonathan Bell and Gail Kaiser. Unit test virtualization with VMVM. In *International Conference on Software Engineering*, pages 550–561. ACM, 2014. `https://doi.org/10.1145/2568225.2568248`.

[13] Jonathan Bell and Luís Pina. CROCHET: Checkpoint and rollback via lightweight heap traversal on stock JVMs. In *Proceedings of the 2018 European Conference on Object-Oriented Programming*, pages 17:1–17:31. Dagstuhl, 2018. `https://doi.org/10.4230/LIPIcs.ECOOP.2018.17`.

[14] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. JIT-picking: Differential fuzzing of JavaScript engines. In

*Conference on Computer and Communications Security*, pages 351–364. ACM, 2022. `https://doi.org/10.1145/3548606.3560624`.

[15] Denis Bogdanas and Grigore Roşu. K-Java: A complete semantics of Java. In *Symposium on Principles of Programming Languages*, pages 445–456. ACM, 2015. `https://doi.org/10.1145/2676726.2676982`.

[16] Tegan Brennan, Nicolás Rosner, and Tevfik Bultan. JIT leaks: Inducing timing side channels through just-in-time compilation. In *Symposium on Security and Privacy*, pages 1207–1222. IEEE, 2020. `https://doi.org/10.1109/SP40000.2020.00007`.

[17] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. JVM fuzzing for JIT-induced side-channel detection. In *International Conference on Software Engineering*, pages 1011–1023. ACM, 2020. `https://doi.org/10.1145/3377811.3380432`.

[18] Fraser Brown, John Renner, Andres Nötzli, Sorin Lerner, Hovav Shacham, and Deian Stefan. Towards a verified range analysis for JavaScript JITs. In *Programming Language Design and Implementation*, pages 135–150. ACM, 2020. `https://doi.org/10.1145/3385412.3385968`.

[19] Stefan Brunthaler. Efficient interpretation using quickening. In *Symposium on Dynamic Languages*, pages 1–14. ACM, 2010. `https://doi.org/10.1145/1869631.1869633`.

[20] Ben Buhse, Thomas Wei, Zhiqiang Zang, Aleksandar Milicevic, and Milos Gligoric. VeDebug: Regression debugging tool for Java. In *International Conference on Software Engineering, Tool Demonstrations Track*, pages 15–18, 2019. `https://doi.org/10.1109/ICSE-Companion.2019.00027`.

[21] Heung Seok Chae, Gyun Woo, Tae Yeon Kim, Jung Ho Bae, and Won-Young Kim. An automated approach to reducing test suites for testing retargeted

C compilers for embedded systems. *Journal of Systems and Software*, 84(12): 2053–2064, 2011. https://doi.org/10.1016/j.jss.2011.04.023.

[22] Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. Finding typing compiler bugs. In *Programming Language Design and Implementation*, pages 183–198. ACM, 2022. https://doi.org/10.1145/3519939.3523427.

[23] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In *International Conference on Software Engineering*, pages 180–190. ACM, 2016. https://doi.org/10.1145/2884781.2884878.

[24] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. History-guided configuration diversification for compiler test-program generation. In *International Conference on Software Engineering*, pages 305–316. IEEE, 2019. https://doi.org/10.1109/ASE.2019.00037.

[25] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *ACM Computing Surveys*, 53(1):4:1–4:36, 2020. https://doi.org/10.1145/3363562.

[26] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. Coverage prediction for accelerating compiler testing. *IEEE Transactions on Software Engineering*, 47(2):261–278, 2021. https://doi.org/10.1109/TSE.2018.2889771.

[27] Ping Chen, Yi Fang, Bing Mao, and Li Xie. JITDefender: A defense against JIT spraying attacks. In *International Conference on ICT Systems Security and Privacy*, pages 142–153. Springer, 2011. https://doi.org/10.1007/978-3-642-21424-0_12.

[28] Ping Chen, Rui Wu, and Bing Mao. JITSafe: a framework against Just-in-time spraying attacks. *IET Information Security*, 7(4):283–292, 2013. `https://doi.org/10.1049/iet-ifs.2012.0142`.

[29] T.Y. Chen, S.C. Cheung, and S.M. Yiu. Metamorphic testing: A new approach for generating next test cases. Technical report, 1998. `https://arxiv.org/abs/2002.12543`.

[30] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Programming Language Design and Implementation*, pages 197–208. ACM, 2013. `https://doi.org/10.1145/2491956.2462173`.

[31] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. One engine to fuzz 'em all: Generic language processor testing with semantic validation. In *Symposium on Security and Privacy*, pages 642–658. IEEE, 2021. `https://doi.org/10.1109/SP40001.2021.00071`.

[32] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of JVM implementations. In *Programming Language Design and Implementation*, pages 85–99. ACM, 2016. `https://doi.org/10.1145/2908080.2908095`.

[33] Yuting Chen, Ting Su, and Zhendong Su. Deep differential testing of JVM implementations. In *International Conference on Software Engineering*, pages 1257–1268. IEEE, 2019. `https://doi.org/10.1109/ICSE.2019.00127`.

[34] Wai-Mee Ching and Alex Katz. The testing of an APL compiler. In *International Conference on APL*, pages 55–62. ACM, 1993. `https://doi.org/10.1145/166197.166205`.

[35] Shafiul Azam Chowdhury, Sohil Lal Shrestha, Taylor T. Johnson, and Christoph Csallner. SLEMI: Equivalence modulo input (EMI) based mutation of CPS

models for finding compiler bugs in Simulink. In *International Conference on Software Engineering*, pages 335–346. ACM, 2020. `https://doi.org/10.1145/3377811.3380381`.

[36] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming*, pages 268–279. ACM, 2000. `https://doi.org/10.1145/351240.351266`.

[37] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *International Symposium on Software Reliability Engineering*, pages 201–211. IEEE, 2014. `https://doi.org/10.1109/ISSRE.2014.11`.

[38] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pages 185–194. ACM, 2007. `https://doi.org/10.1145/1287624.1287651`.

[39] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. `https://doi.org/10.1007/978-3-540-78800-3_24`.

[40] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978. `https://doi.org/10.1109/C-M.1978.218136`.

[41] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 93:1–93:29. ACM, 2017. `https://doi.org/10.1145/3133917`.

[42] Eclipse Foundation, Inc. JDK 8/11/17/18 - wrong result from JIT compilation with modular operation and field access, 2022. `https://github.com/eclipse-openj9/openj9/issues/17129`.

[43] Eclipse Foundation, Inc. Eclipse IDE | the Eclipse Foundation, 2023. `https://eclipseide.org`.

[44] Eclipse Foundation, Inc. The order of super interface initialization in J9 is strange - issue #13242 - eclipse-openj9/openj9, 2024. `https://github.com/eclipse-openj9/openj9/issues/13242`.

[45] Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. CsmithEdge: more effective compiler testing by handling undefined behaviour less conservatively. *Empirical Software Engineering*, 27(6), 2022. `https://doi.org/10.1007/s10664-022-10146-1`.

[46] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. GrayC: Greybox fuzzing of compilers and analysers for C. In *International Symposium on Software Testing and Analysis*, pages 1219–1231. ACM, 2023. `https://doi.org/10.1145/3597926.3598130`.

[47] Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. Correctness of speculative optimizations with dynamic deoptimization. In *Symposium on Principles of Programming Languages*, pages 49:1–49:28. ACM, 2017. `https://doi.org/10.1145/3158137`.

[48] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pages 416–419. ACM, 2011. `https://doi.org/10.1145/2025113.2025179`.

[49] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. JITGuard: Hardening just-in-time compilers with SGX. In *Conference on Computer and Communications Security*, pages 2405–2419. ACM, 2017. `https://doi.org/10.1145/3133956.3134037`.

[50] Free Software Foundation, Inc. Testsuites (GNU compiler collection (GCC) internals), 2021. `https://gcc.gnu.org/onlinedocs/gccint/Testsuites.html`.

[51] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. CodeHint: Dynamic and interactive synthesis of code snippets. In *International Conference on Software Engineering*, pages 653–663. ACM, 2014. `https://doi.org/10.1145/2568225.2568250`.

[52] Tianchang Gao, Junjie Chen, Yingquan Zhao, Yuqun Zhang, and Lingming Zhang. Vectorizing program ingredients for better JVM testing. In *International Symposium on Software Testing and Analysis*, pages 526–537. ACM, 2023. `https://doi.org/10.1145/3597926.3598075`.

[53] Robert Gawlik and Thorsten Holz. SoK: Make JIT-Spray great again. In *USENIX Workshop on Offensive Technologies*. USENIX, 2018. `https://www.usenix.org/conference/woot18/presentation/gawlik`.

[54] GitHub, Inc. Github, 2023. `https://github.com`.

[55] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in UDITA. In *International Conference on Software Engineering*, pages 225–234. ACM, 2010. `https://doi.org/10.1145/1806799.1806835`.

[56] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Programming Language Design and Implementation*, pages 213–223. ACM, 2005. `https://doi.org/10.1145/1065010.1065036`.

[57] James Gosling and Greg Bollella. *The Real-Time Specification for Java*. Addison-Wesley, 2000. ISBN 0201703238.

[58] Samuel Groß. FuzzIL: Coverage guided fuzzing for JavaScript engines, 2018. `https://api.semanticscholar.org/CorpusID:228320904`.

[59] Shu-yu Guo and Jens Palsberg. The essence of compiling with traces. In *Symposium on Principles of Programming Languages*, pages 563–574. ACM, 2011. `https://doi.org/10.1145/1925844.1926450`.

[60] Alex Gyori, Ben Lambeth, August Shi, Owolabi Legunsen, and Darko Marinov. NonDex: A tool for detecting and debugging wrong assumptions on Java API specifications. In *International Symposium on the Foundations of Software Engineering*, pages 993–997. ACM, 2016. `https://doi.org/10.1145/2950290.2983932`.

[61] Andrew Haley. How to change compilation policy to trigger C2 compilation ASAP?, 2015. `https://mail.openjdk.java.net/pipermail/hotspot-compiler-dev/2015-May/018010.html`.

[62] Chris Hawblitzel, Shuvendu K. Lahiri, Kshama Pawar, Hammad Hashmi, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, and Scott Wadsworth. Will you still compile me tomorrow? static cross-version compiler validation. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pages 191–201. ACM, 2013. `https://doi.org/10.1145/2491411.2491442`.

[63] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *USENIX Security Symposium*, pages 445–458. USENIX, 2012. `https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler`.

[64] Josie Holmes and Alex Groce. Causal distance-metric-based assistance for debugging after compiler fuzzing. In *International Symposium on Software Reliability Engineering*, pages 166–177. IEEE, 2018. https://doi.org/10.1109/ISSRE.2018.00027.

[65] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Librando: Transparent code randomization for Just-in-Time compilers. In *Conference on Computer and Communications Security*, pages 993–1004. ACM, 2013. https://doi.org/10.1145/2508859.2516675.

[66] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A domain-specific language for building self-optimizing ast interpreters. In *International Conference on Generative Programming: Concepts and Experiences*, pages 123–132. ACM, 2014. https://doi.org/10.1145/2775053.2658776.

[67] Radu Iosif. Symmetry reduction criteria for software model checking. In *International SPIN Symposium on Model Checking of Software*, pages 22–41. Springer, 2002. https://doi.org/10.1007/3-540-46017-9_5.

[68] JavaParser.org. Javaparser - home, 2024. https://javaparser.org.

[69] Haoxiang Jia, Ming Wen, Zifan Xie, Xiaochen Guo, Rongxin Wu, Maolin Sun, Kang Chen, and Hai Jin. Detecting JVM JIT compiler bugs via exploring two-dimensional input spaces. In *International Conference on Software Engineering*, pages 43–55. IEEE, 2023. https://doi.org/10.1109/ICSE48619.2023.00016.

[70] Yue Jia and Mark Harman. Constructing subtle faults using higher order mutation testing. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 249–258. IEEE, 2008. https://doi.org/10.1109/SCAM.2008.36.

[71] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993. ISBN 9780130202499.

[72] JUnit. JUnit - about, 2022. `https://junit.org/junit4/`.

[73] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. `https://doi.org/10.1145/360248.360252`.

[74] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2018. `https://doi.org/10.1145/3243734.3243804`.

[75] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Symposium on Principles of Programming Languages*, pages 179–191. ACM, 2014. `https://doi.org/10.1145/2578855.2535841`.

[76] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Programming Language Design and Implementation*, pages 216–226. ACM, 2014. `https://doi.org/10.1145/2666356.2594334`.

[77] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 386–399. ACM, 2015. `https://doi.org/10.1145/2858965.2814319`.

[78] Xavier Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009. `https://doi.org/10.1145/1538788.1538814`.

[79] Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[80] Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. Validating JIT compilers via compilation space exploration. In *Symposium on Operating Systems*

*Principles*, pages 66–79. ACM, 2023. https://doi.org/10.1145/3600006.3613140.

[81] Wilson Lian, Hovav Shacham, and Stefan Savagem. Too LeJIT to quit: Extending JIT spraying to ARM. In *The Symposium on Network and Distributed System Security*. The Internet Society, 2015. https://doi.org/10.14722/ndss.2015.23288.

[82] Wilson Lian, Hovav Shacham, and Stefan Savage. A call to ARMs: Understanding the costs and benefits of JIT spraying mitigations. In *The Symposium on Network and Distributed System Security*. The Internet Society, 2017. https://doi.org/10.14722/NDSS.2017.23108.

[83] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. In *Programming Language Design and Implementation*, pages 65–76. ACM, 2015. https://doi.org/10.1145/2737924.2737986.

[84] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for C and C++ compilers with YARPGen. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 196:1–196:25. ACM, 2020. https://doi.org/10.1145/3428264.

[85] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Fuzzing loop optimizations in compilers for C++ and data-parallel languages. In *PLDI*. ACM, 2023. https://doi.org/10.1145/3591295.

[86] LLVM Project. LLVM testing infrastructure guide, 2021. https://llvm.org/docs/TestingGuide.html.

[87] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998. https://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf.

[88] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *Programming Language Design and Implementation*, pages 187–196. ACM, 2013. https://doi.org/10.1145/2491956.2491967.

[89] Mozilla Foundation. Mozillasecurity/funfuzz: A collection of fuzzers in a harness for testing the SpiderMonkey JavaScript engine., 2023. https://github.com/MozillaSecurity/funfuzz.

[90] Kazuhiro Nakamura and Nagisa Ishiura. Random testing of C compilers based on test program generation by equivalence transformation. In *Asia Pacific Conference on Circuits and Systems*, pages 676–679. IEEE, 2016. https://doi.org/10.1109/APCCAS.2016.7804063.

[91] Jakob Botsch Nielsen. Fuzzing the .NET JIT compiler, 2018. https://mattwarren.org/2018/08/28/Fuzzing-the-.NET-JIT-Compiler/.

[92] Jakob Botsch Nielsen. jakobbotsch/Fuzzlyn: Fuzzer for the .NET toolchains, developed as a project for the 2018 language-based security course at Aarhus University, 2020. https://github.com/jakobbotsch/Fuzzlyn.

[93] Ben Niu and Gang Tan. RockJIT: Securing Just-In-Time compilation using modular control-flow integrity. In *Conference on Computer and Communications Security*, pages 1317–1328. ACM, 2014. https://doi.org/10.1145/2660267.2660281.

[94] Oracle Corporation and/or its affiliates. The Java HotSpot performance engine architecture, 2021. https://www.oracle.com/java/technologies/whitepaper.html.

[95] Oracle Corporation and/or its affiliates. [GR-45498] incorrect result from Graal compilation with String.getBytes(Charset), 2022. https://github.com/oracle/graal/issues/6403.

[96] Oracle Corporation and/or its affiliates. Regression test harness for the JDK: jtreg, 2022. https://openjdk.java.net/jtreg.

[97] Oracle Corporation and/or its affiliates. [JDK-8280126] C2: detect and remove dead irreducible loops - java bug system, 2023. https://bugs.openjdk.java.net/browse/JDK-8280126.

[98] Oracle Corporation and/or its affiliates. [JDK-8280126] C2: detect and remove dead irreducible loops - java bug system, 2023. https://bugs.openjdk.org/browse/JDK-8280126?focusedCommentId=14476253&page=com.atlassian.jira.plugin.system.issuetabpanels%3Acomment-tabpanel#comment-14476253.

[99] Oracle Corporation and/or its affiliates. openjdk/jdk: JDK main-line development, 2023. https://github.com/openjdk/jdk.

[100] OW2. Asm, 2024. https://asm.ow2.io.

[101] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, pages 75–84. IEEE, 2007. https://doi.org/10.1109/ICSE.2007.37.

[102] Rohan Padhye, Caroline Lemieux, and Koushik Sen. JQF: Coverage-guided property-based testing in Java. In *International Symposium on Software Testing and Analysis*, pages 398–401. ACM, 2019. https://doi.org/10.1145/3293882.3339002.

[103] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with Zest. In *International Symposium on Software Testing and Analysis*, pages 329–340. ACM, 2019. https://doi.org/10.1145/3293882.3330576.

[104] Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu. JEST: N+1-version differential testing of both JavaScript engines and specification. In *International Conference on Software Engineering*, pages 13–24. IEEE, 2021. https://doi.org/10.1109/ICSE43902.2021.00015.

[105] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing JavaScript engines with aspect-preserving mutation. In *Symposium on Security and Privacy*, pages 1629–1642. IEEE, 2020. https://doi.org/10.1109/SP40000.2020.00067.

[106] Guillermo Polito, Stéphane Ducasse, and Pablo Tesone. Interpreter-guided differential JIT compiler unit testing. In *Programming Language Design and Implementation*, pages 981–992. ACM, 2022. https://doi.org/10.1145/3519939.3523457.

[107] Priya Khaira-Hanks. What is a Java Uber-JAR and why is it useful?, 2023. https://blog.payara.fish/what-is-a-java-uber-jar.

[108] Qi Qin, JulianAndres JiYang, Fu Song, Taolue Chen, and Xinyu Xing. DeJITLeak: Eliminating JIT-induced timing side-channel leaks. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pages 872–884. ACM, 2022. https://doi.org/10.1145/3540250.3549150.

[109] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *Programming Language Design and Implementation*, pages 335–346. ACM, 2012. https://doi.org/10.1145/2345156.2254104.

[110] Richard Schumi and Jun Sun. SpecTest: Specification-based compiler testing. In *Fundamental Approaches to Software Engineering*, pages 269–291. Springer, 2021. https://doi.org/10.1007/978-3-030-71500-7_14.

[111] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pages 263–272. ACM, 2005. `https://doi.org/10.1145/1081706.1081750`.

[112] Rohan Sharma, Milos Gligoric, Andrea Arcuri, Gordon Fraser, and Darko Marinov. Testing container classes: Random or systematic? In *FASE*, pages 262–277. Springer, 2011. `https://doi.org/10.1007/978-3-642-19811-3_19`.

[113] Boris Shingarov. Formal verification of JIT by symbolic execution. In *International Workshop on Virtual Machines and Intermediate Languages*, 2019.

[114] Emin Gün Sirer and Brian N. Bershad. Using production grammars in software testing. In *Conference on Domain-Specific Languages*, pages 1–13. ACM, 2000. `https://doi.org/10.1145/331960.331965`.

[115] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5–6):475–495, 2013. `https://doi.org/10.1007/s10009-012-0249-7`.

[116] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 404–415. ACM, 2006. `https://doi.org/10.1145/1168857.1168907`.

[117] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. Exploiting and protecting dynamic code generation. In *The Symposium on Network and Distributed System Security*. The Internet Society, 2015. `https://doi.org/10.14722/NDSS.2015.23233`.

[118] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *International Conference on Object-Oriented Programming,*

*Systems, Languages, and Applications*, pages 849–863. ACM, 2016. https://doi.org/10.1145/2983990.2984038.

[119] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided program reduction. In *International Conference on Software Engineering*, pages 361–371. ACM, 2018. https://doi.org/10.1145/3180155.3180236.

[120] Yixuan Tang, Zhilei Ren, Weiqiang Kong, and He Jiang. Compiler testing: a systematic literature analysis. *Frontiers of Computer Science*, 14(1):1:20, 2020. https://doi.org/10.1007/s11704-019-8231-0.

[121] The Apache Software Foundation. Maven - welcome to Apache Maven, 2023. https://maven.apache.org/.

[122] The Apache Software Foundation. commons-codec/BinaryCodec.java, 2023. https://github.com/apache/commons-codec/blob/4de60e/src/main/java/org/apache/commons/codec/binary/BinaryCodec.java.

[123] The Apache Software Foundation. commons-codec/StringUtils.java, 2023. https://github.com/apache/commons-codec/blob/4de60e/src/main/java/org/apache/commons/codec/binary/StringUtils.java.

[124] The Apache Software Foundation. commons-math/AdamsNordsieckTransformer.java, 2023. https://github.com/apache/commons-math/blob/dff1a0/src/main/java/org/apache/commons/math4/ode/nonstiff/AdamsNordsieckTransformer.java.

[125] The Apache Software Foundation. commons-math/Quaternion.java, 2023. https://github.com/apache/commons-math/blob/dff1a0/src/main/java/org/apache/commons/math4/complex/Quaternion.java.

[126] The Apache Software Foundation. commons-text/StrBuilder.java, 2023. `https://github.com/apache/commons-text/blob/e62203/src/main/java/org/apache/commons/text/StrBuilder.java`.

[127] The JUnit Team. JUnit 5, 2023. `https://junit.org/junit5/`.

[128] The MITRE Corporation. CVE - CVE-2020-14792, 2022. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-14792`.

[129] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pages 253–262. ACM, 2005. `https://doi.org/10.1145/1095430.1081749`.

[130] Jacob Van Geffen, Luke Nelson, Isil Dillig, Xi Wang, and Emina Torlak. Synthesizing JIT compilers for in-kernel DSLs. In *Computer Aided Verification*, pages 564–586. Springer, 2020. `https://doi.org/10.1007/978-3-030-53291-8_29`.

[131] Vasudev Vikram, Rohan Padhye, and Koushik Sen. Growing a test corpus with Bonsai fuzzing. In *International Conference on Software Engineering*, pages 723–735. ACM, 2021. `https://doi.org/10.1109/ICSE43902.2021.00072`.

[132] Willem Visser, Corina S. Pundefinedsundefinedreanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *International Symposium on Software Testing and Analysis*, pages 97–107. ACM, 2004. `https://doi.org/10.1145/1007512.1007526`.

[133] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. FuzzJIT: Oracle-enhanced fuzzing for JavaScript engine JIT compiler. In *USENIX Security Symposium*, pages 1865–1882. USENIX, 2023. `https://www.usenix.org/conference/usenixsecurity23/presentation/wang-junjie`.

[134] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *Symposium on Operating Systems Design and Implementation*, pages 33–47. USENIX, 2014. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/wang_xi.

[135] Tao Wei, Tielei Wang, Lei Duan, and Jing Luo. INSeRT: Protect dynamic code generation against spraying. In *International Conference on Information Science and Technology*, pages 323–328. IEEE, 2011. https://doi.org/10.1109/ICIST.2011.5765261.

[136] Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. JITfuzz: Coverage-guided fuzzing for JVM Just-in-Time compilers. In *International Conference on Software Engineering*, pages 56–68. IEEE, 2023. https://doi.org/10.1109/ICSE48619.2023.00017.

[137] Mingyuan Wu, Yicheng Ouyang, Minghai Lu, Junjie Chen, Yingquan Zhao, Heming Cui, Guowei Yang, and Yuqun Zhang. SJFuzz: Seed & mutator scheduling for JVM fuzzing. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pages 1062–1074. ACM, 2023. https://doi.org/10.1145/3611643.3616277.

[138] Rui Wu, Ping Chen, Bing Mao, and Li Xie. RIM: A method to defend from JIT spraying attack. In *International Conference on Availability, Reliability and Security*, pages 143–148. IEEE, 2012. https://doi.org/10.1109/ARES.2012.11.

[139] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Programming Language Design and Implementation*, pages 283–294. ACM, 2011. https://doi.org/10.1145/1993316.1993532.

[140] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. GCC bug list found by random testing (total 79), 2021. https://embed.cs.utah.edu/csmith/gcc-bugs.html.

[141] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. LLVM bug list found by random testing (total 203), 2021. https://embed.cs.utah.edu/csmith/llvm-bugs.html.

[142] Guixin Ye, Tianmin Hu, Zhanyong Tang, Zhenye Fan, Shin Tan, Hwei, Bo Zhang, Wenxiang Qian, and Wang Zheng. A generative and mutational approach for synthesizing bug-exposing test cases to guide compiler fuzzing. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pages 1127–1139. ACM, 2023. https://doi.org/10.1145/3611643.3616332.

[143] Takahide Yoshikawa, Kouya Shimura, and Toshihiro Ozawa. Random program generator for Java JIT compiler test system. In *International Conference on Quality Software*, pages 20–23. IEEE, 2003. https://doi.org/10.1109/QSIC.2003.1319081.

[144] Zhiqiang Zang, Nathaniel Wiatrek, Milos Gligoric, and August Shi. Compiler testing using template Java programs. In *International Conference on Automated Software Engineering*, pages 23:1–23:13. ACM, 2022. https://doi.org/10.1145/3551349.3556958.

[145] Zhiqiang Zang, Aditya Thimmaiah, and Milos Gligoric. Pattern-based peephole optimizations with Java JIT tests. In *International Symposium on Software Testing and Analysis*, pages 64–75, 2023. https://doi.org/10.1145/3597926.3598038.

[146] Zhiqiang Zang, Fu-Yao Yu, Nathaniel Wiatrek, Milos Gligoric, and August Shi. JAttack: Java JIT testing using template programs. In *International Confer-*

*ence on Software Engineering, Tool Demonstrations Track*, pages 6–10. IEEE, 2023. https://doi.org/10.1109/ICSE-Companion58688.2023.00014.

[147] Zhiqiang Zang, Aditya Thimmaiah, and Milos Gligoric. JOG: Java JIT peephole optimizations and tests from patterns. In *International Conference on Software Engineering, Tool Demonstrations Track*, page to apear, 2024. https://doi.org/10.1145/3639478.3640040.

[148] Zhiqiang Zang, Fu-Yao Yu, Aditya Thimmaiah, August Shi, and Milos Gligoric. Java JIT testing with template extraction. In *International Symposium on the Foundations of Software Engineering*, page to appear. ACM, 2024. https://doi.org/10.1145/3643777.

[149] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pages 253–267. Springer, 1999. https://doi.org/10.1145/318774.318946.

[150] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002. https://doi.org/10.1109/32.988498.

[151] Chao Zhang, Mehrdad Niknami, Kevin Zhijie Chen, Chengyu Song, Zhaofeng Chen, and Dawn Song. JITScope: Protecting web users from control-flow hijacking attacks. In *International Conference on Computer Communications*, pages 567–575. IEEE, 2015. https://doi.org/10.1109/INFOCOM.2015.7218424.

[152] Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In *Programming Language Design and Implementation*, pages 347–361. ACM, 2017. https://doi.org/10.1145/3140587.3062379.

[153] Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. History-driven test program synthesis for JVM testing. In *International Conference on Software Engineering*, pages 1133–1144. ACM, 2022. `https://doi.org/10.1145/3510003.3510059`.