Validating JIT Compilers via Compilation Space Exploration

CONG LI, State Key Laboratory for Novel Software Technology, Nanjing University, China
YANYAN JIANG, State Key Laboratory for Novel Software Technology, Nanjing University, China
CHANG XU, State Key Laboratory for Novel Software Technology, Nanjing University, China
ZHENDONG SU, Department of Computer Science, ETH Zurich, Switzerland

We introduce the concept of *compilation space* as a new pivot for the comprehensive validation of just-in-time (JIT) compilers in modern language virtual machines (LVMs). The compilation space of a program, encompasses a wide range of equivalent JIT-compilation choices, which can be cross-validated to ensure the correctness of the program's JIT compilations. To thoroughly explore the compilation space in a lightweight and LVM-agnostic manner, we strategically mutate test programs with JIT-relevant but semantics-preserving code constructs, aiming to provoke diverse JIT compilation optimizations. We primarily implement this approach in `Artemis`, a tool for validating Java Virtual Machines (JVMs). Within three months, `Artemis` successfully discovered 85 bugs in three widely used production JVMs — HotSpot, OpenJ9, and the Android Runtime — where 53 were already confirmed or fixed and many of which were classified as critical. It is noteworthy that all reported bugs concern JIT compilers, highlighting the effectiveness and practicality of our technique. Building on the promising results with JVMs, we experimentally applied our technique to a state-of-the-art JavaScript Engine (JSE) fuzzer called Fuzzilli, aiming to augment it to find mis-compilation bugs without significantly sacrificing its ability to detect crashes. Our experiments demonstrate that our enhanced version of Fuzzilli namely `Apollo` could achieve comparable code coverage with a considerably smaller number of generated programs with a similar number of crashes. Additionally, `Apollo` successfully uncovered four mis-compilations in JavaScriptCore and SpiderMonkey within seven days. Following `Artemis`' and `Apollo`'s success, we are expecting that the generality and practicability of our approach will make it broadly applicable for understanding and validating the JIT compilers of other LVMs.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; *Dynamic compilers*; *Interpreters*; *Virtual machines*; • **Computer systems organization → Reliability**.

Additional Key Words and Phrases: Just-in-time compilers, Java virtual machines, JavaScript engines, testing

ACM Reference Format:
Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. 2025. Validating JIT Compilers via Compilation Space Exploration. *ACM Trans. Comput. Syst.* 1, 1, Article 1 (January 2025), 37 pages. https://doi.org/10.1145/3715102

1 Introduction

Modern programming language virtual machines (LVMs) are among the most critical and widely used systems software ever developed. These LVMs typically *interleave* straightforward bytecode interpretation with dynamic, just-in-time (JIT) compilation for improved performance. Well-known JIT compilers include HotSpot's C1 and C2 compilers, V8's Turbofan and Maglev engines, and the JIT compiler of Linux eBPF. Aligning with modern compilers, JIT compilers implement a

Cong Li (congli@smail.nju.edu.cn), Yanyan Jiang (corresponding, jyy@nju.edu.cn), and Chang Xu (changxu@nju.edu.cn) are with the State Key Laboratory for Novel Software Technology at Nanjing University, Nanjing, China; Zhendong Su (zhendong.su@inf.ethz.ch) is with the Department of Computer Science at ETH Zurich, Zurich, Switzerland.

ACM 1557-7333/2025/1-ART1
https://doi.org/10.1145/3715102

```
class T {
  int baz() { return 1; }
  int bar() { return 2; }
  int foo() {
    return bar() + baz();
  }
  int main() {
    return foo();
  }
}
```

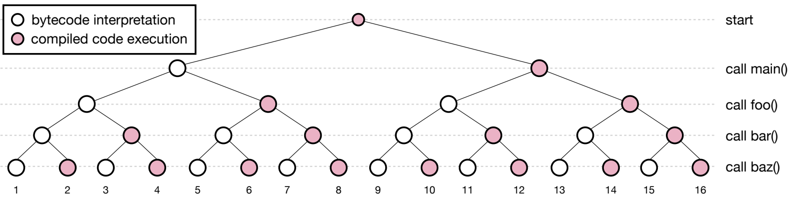

Fig. 1. The compilation space of the program on the left, assuming that (1) each method call can be independently compiled or interpreted and (2) on-stack replacement and background compilations are disabled.

broad range of non-trivial optimizations such as loop unrolling. The interleaving and all involved optimizations make them one of the most complex components of modern LVMs, being one of the primary sources of bugs. For example, a recent report indicates that approximately 45% of V8's Common Vulnerabilities and Exposures (CVEs) after 2019 are related to JIT compilers [43]. As a result, both industry and academia have made significant efforts to ensure the correctness of LVMs [10, 22, 24, 81], especially their JIT compilers [3, 20, 67, 71].

However, validating JIT compilers is challenging. On one hand, generating test cases with valid syntax and semantics is notoriously difficult [10, 11, 24, 44, 83]. This difficulty often results in a limited number of tests being able to reach the baseline JIT compiler that is typically warmed up through a lengthy profiling process. The tiered nature of JIT compilation [26, 62, 69, 70] further makes it even harder to engage with the more advanced, optimizing JIT compilers. On the other hand, even for tests that successfully reach the JIT compilers, exploiting various JIT optimizations poses another layer of difficulties [46, 52, 53, 81]. Consequently, most identified bugs are superficial and often unrelated to JIT compilers, typically stemming from early-stage parser or verifier issues.

To tackle these challenges, this paper presents a novel approach to finding deep *JIT-compiler bugs* — specifically crashes and mis-compilations — that do *not* manifest in the LVM's interpretation mode. Unlike existing techniques that consider JIT compilers as *static* compilers [1, 3, 29, 44, 67], our approach thoroughly exploits the *dynamic* interleaving between bytecode interpretation and compiled-code execution. This feature renders our approach the first — as of the publication time of the conference version of this paper on SOSP 2023 [33] — work to validate JIT compilers through their dynamic nature, to the best of our knowledge.

Compilation Space modulo LVM. The key idea of our approach is to model the interleaving behavior between an LVM's interpreter and its JIT compilers by the concept of *compilation space modulo LVM* (or compilation space for short), a space of equivalent JIT-compilation choices.

Assuming that a program P makes n method[1] calls at runtime where each method call can be independently *compiled* or *interpreted*, we get a compilation space consisting of 2^n possible JIT-compilation choices. These choices naturally correspond to 2^n program versions, where each version can be conceptually regarded as the program P plus a specific JIT-compilation choice. Additionally, the space indicates a considerably strong *test oracle* relating to equivalence, as the execution of any version of the program should consistently produce the same output. The concept of compilation space significantly extends the testing space (implicitly) utilized by traditional approaches, which only consider a limited number of JIT-compilation choices.

Figure 1 depicts the compilation space of the program displayed on the left. The program is as simple as making 4 method calls, resulting in a total of 2^4 equivalent JIT-compilation choices within its compilation space. However, traditional approaches [29, 67] only consider certain choices, such as the fully interpreted choice (#1) and the fully compiled choice (#16), leaving the remainder

[1] Our descriptions throughout the paper adhere to an object-oriented style, employing terms such as classes and methods.

unexplored. Notably, the program in this example should always return 3 for the `main()` method, regardless of which JIT-compilation choice ranging from #1 to #16 is executed. Any violation of this assertion would indicate a bug in an LVM's JIT compiler, thus serving as our test oracle.

In fact, the simplistic two-state (i.e., independently compiled or interpreted as aforementioned) assumption is an oversimplification. In practical LVMs such as HotSpot, the transition between interpretation and compilation (or the other way around) is far more complicated, which could occur multiple times within the body of a method rather than being limited to once at the method boundary. Consequently, the actual compilation space is significantly larger than 2^n, providing even more opportunities for JIT-compiler validation. We will detail further background on JIT compilers in Section 2 and formalize the compilation space in Section 3.

Compilation Space Exploration. Another advantage of the compilation space is that different JIT-compilation choices are likely to trigger distinct optimization paths and passes within a JIT compiler. This inherent diversity makes it beneficial to systematically explore every possible choice within the space and to cross-validate the output equivalence between each pair of choices. In this paper, we refer to this novel approach as *Compilation Space Exploration* (CSX).

A straightforward and idealized realization of CSX is modifying the LVM under validation to expose an interface for free control of the LVM's dynamic interleaving. However, such modifications are likely to render it incompatible with the LVM's internal assumptions. The modifications also evidently require substantial engineering effort as there are many LVM implementations.

To make CSX applicable to a wide range of JIT compilers, we propose *JIT-Op Neutral Mutation* (JoNM). This LVM-agnostic strategy manages the LVM's decisions on switching between interpretation and JIT-compilation mode through lightweight source-level mutations, rather than extensive modifications at the LVM level. Our key observation is that CSE can be approximated by leveraging the mechanism of profile-guided speculative compilation/optimization: Modern JIT compilers profile the program to identify and compile "hot code" (i.e., frequently executed segments) while leaving "uncommon traps" that allow them to fall back to the bytecode interpreter in the event of failed speculations. Therefore, we can either insert/remove loops or method calls to enable/disable JIT compilation, or manipulate uncommon cold paths to control whether the LVM falls back to interpretation. However, it is essential to carefully design these mutations to ensure they do not alter the semantics of the modified program, allowing us to utilize the test oracle for cross-validation; this is why we call them neutral (i.e., semantics-preserving) mutations.

JoNM enables us to interleave between interpretation and compilation, approximating exploring the compilation space. For example, we can potentially shift from choice #1 to choice #6 in Figure 1 by inserting semantics-preserving loops within the `foo()` and `baz()` methods. We will introduce more details of Compilation Space Exploration and JIT-Op Neutral Mutation in Section 3.

Implementations. We primarily developed 🦌`Artemis`, an implementation of JoNM that only manipulates method calls and loops for validating Java Virtual Machines (JVMs). Leveraging `Artemis`, we tested three widely used production JVMs — HotSpot, OpenJ9, and the Android Runtime (ART) — and finally reported 85 JIT-compiler bugs to them. Among these bugs, 53 have been confirmed/fixed and many were critical: 12 OpenJ9 bugs were classified as `blocker`, the most severe, release-blocking type of bugs; 10 HotSpot bugs were marked as at least `P3`, major loss of function; 13 OpenJ9 bugs were long latent across ≥4 major and many minor releases. Our reported bugs stem from diverse root causes in various JIT-compiler components, such as global value numbering, loop optimization, and code generation. We also received positive feedback from the respective JVM developers like *"I noticed that you filed quite a few bug reports for the JITs recently, thanks a lot for that ...I'm looking forward to learning more about your research."* The evaluations of `Artemis` are presented in Section 4.

Beyond that, we experimentally applied JoNM to augment Fuzzilli [20] with the ability to find mis-compilations in JavaScript Engines (JSEs). We thereby built 🦅`Apollo` based on Fuzzilli. Our experiments on top of seven popular JSEs — QuickJS, JerryScript, Duktape, Nginx JavaScript, V8, JavaScriptCore, and SpiderMonkey — show that `Apollo` could achieve comparable or even superior code coverage with significantly fewer generated programs while still leading to a similar number of crashes compared with Fuzzilli. When applying `Apollo` in a seven-day fuzzing campaign, we successfully uncovered four mis-compilations in JavaScriptCore and SpiderMonkey. The evaluations of `Apollo` are elaborated on in Section 5.

Contributions. Overall, we make the following contributions:

- We present the concept of compilation space and the novel CSX approach for thorough validation of JIT compilers. We propose JoNM, an LVM-agnostic strategy to effectively approximate CSX via program mutations at the source code level.
- We show that even a basic JoNM implementation can unleash CSX's power: 85 JIT-compiler bugs are discovered in three widely used production JVMs, i.e., HotSpot, OpenJ9, and ART.
- We demonstrate the generality and practicability of our approach on other LVMs by applying it to JSEs: We achieve comparable code coverage by generating fewer programs and discover four JIT-compiler mis-compilations in seven days.
- We make `Artemis` and `Apollo` publicly available via the following links to benefit the community and facilitate future research:
    - 🦌`Artemis`. https://github.com/test-jitcomp/Artemis.
    - 🦅`Apollo`. https://github.com/test-jitcomp/Apollo.

2 Background and Illustrative Examples

This section provides background on JIT compilers and offers three concrete examples to illustrate how CSX and JoNM operate: one for OpenJ9, another for HotSpot, and a third for SpiderMonkey.

2.1 Just-in-Time Compilation

JIT compilation is costly. As a result, an LVM must carefully balance the costs associated with JIT compilation against its performance benefits. To ensure a fast start-up, an LVM typically boots in the interpretation mode and incrementally compiles the program that it is executing as it runs [8, 12, 28, 42]. Specifically, the LVM monitors the program's control flow — such as method calls and loop back-jumps — using profiling counters. When a counter surpasses a predefined threshold, indicating that the corresponding bytecode segment is *hot*, the LVM triggers background JIT compilation to compile it into machine code. The unit of JIT compilation can be either a method or another type of code block such as a loop. JIT compilers that focus exclusively on methods are referred to as method-JITs, such as HotSpot's C1 compiler and ART's OptimizingCompiler. In contrast, those that support both methods and other code blocks are known as tracing-JITs, for example, HotSpot's C2 compiler and V8's Turbofan engine. To execute the machine code compiled from JIT compilers, modern LVMs support *On-Stack Replacement* (OSR) to replace the interpreter stack frame with a native stack frame [17]. OSR could happen at the method boundary or within the method body depending on LVM and JIT-compiler implementations. After stack replacement, the execution can continue safely. This process is often termed OSR compilation.

To optimize performance, JIT compilers may make *speculative* assumptions based on the program's profiled behavior. For instance, a JIT compiler might assume that a code block is unreachable if the profiler observes no executions of that block over a prolonged period, and thereby the block does not require compilation. For LVMs with dynamic types such as JSEs, JIT compilers frequently make speculations over the profiled types of certain variables. Whenever the LVM is about to execute JIT-compiled code, the assumptions are re-checked. If all assumptions still hold, the LVM executes the JIT-compiled code safely. Conversely, if any assumption is violated, the program hits an *uncommon trap*, causing the LVM to fall back to the interpreter. This process is usually called *de-optimization* [25]. There is also the possibility that the de-optimized bytecode may become "hot" again, prompting the LVM to reuse its machine code in the code cache or completely *re-compile* it.

Additionally, LVMs often support leveled or tiered optimization, where more aggressive optimizations are applied to already compiled code if it becomes rather hotter [26, 49, 62, 69]. However, more aggressive JIT compilers typically take longer to complete compilation, resulting in the production of more performant machine code. Therefore, these compilers are activated on and more suitable for long-serving code. While not universally true, baseline JIT compilers are predominantly method-JITs, with optimizing JIT compilers being tracing-JITs.

All these mechanisms — while different LVM implementations may have their own assumptions and design their own policies for JIT/OSR compilation, de-optimization, re-compilation, and tiered compilation — point to the fact that an LVM implementation frequently switches back and forth between the bytecode interpreter and its JIT compilers when executing a program.

2.2 Illustrative Examples

Given a seed program, JoNM proposes inserting or removing neutral (i.e., semantics-preserving) program constructs, such as loops, method calls, and uncommon traps, to create equivalent mutants with varying JIT compilation choices. Figure 2 and Figure 3 illustrate two mutants that respectively trigger mis-compilations in OpenJ9 and HotSpot. In both instances, we utilized JavaFuzzer [21] to generate their seed programs and Artemis to derive the highlighted mutants. Additionally, Figure 4 depicts a mutant that crashes SpiderMonkey. In this case, the seed program was generated with Fuzzilli [20] and mutations were applied using Apollo. Due to the complexity and the large size of the original mutants, we only present their simplified versions for brevity.

```
1  class C {
2    long a = 0;
3    int b() {
4      int d = 2, e = 10;
5      for (int m = 7; m < 149; ++m)
6        for (int c = 1; c < 4; c++)
7          for (int n = 1; n < 2; ++n)
8            d += e;
9      return d;
10   }
11   void f() {
12     for (int w = 0; w < 6361; w++)
13       a = b();
14   }
15   public static void main(String[] g) {
16     C t = new C();
17     for (int h = 1; h < 212; h++)
18       t.f();
19     System.out.println(t.a);
20   }
21 }
```

Fig. 2. Issue-15306 triggers a mis-compilation in OpenJ9. JavaFuzzer generates the seed while Artemis inserts the highlighted code. The mutant is reduced by Perses and C-Reduce, and manually cleaned up then simplified for shown.

OpenJ9 Issue-15306. Figure 2 triggers a mis-compilation in OpenJ9. Artemis identified the bug in OpenJ9 version 0.32 (revision 3d06b2f9c, based on OpenJDK 1.8.0_342), but it can also be reproduced in versions as far back as 0.24 with JDK 11. The developers promptly marked it as a blocker — the most severe type of bug that prevents a release.

The seed program assigns C.a a value repeatedly returned by C.b() (Line 13). Since C.b() has no side effects, it should consistently return the fixed integer 4,262 — the value of d at Line 9 — resulting in the program outputting 4,262 as well at Line 19. In the seed program, the method C.b() is finally JIT-compiled at the warm level as its method counter reaches OpenJ9's compilation threshold for that level, while all other methods remain primarily interpreted until the program exits. Artemis alters the JIT-compilation choice by inserting a loop into Line 12. This insertion triggers an additional JIT compilation of C.f() at the warm level, along with two additional OSR compilations of the inserted loop at the `veryHot` and `scorching` levels, respectively. Importantly, the inserted loop is neutral and does not change the semantics of the seed program, as `C.b()` continues to return 4,262. Therefore, the mutant should also output 4,262 at Line 19. However, OpenJ9's JIT compiler mis-compiles the mutant, yielding an output of 1,422.

```
 1 class C {
 2   boolean z = false;
 3   byte l = 0;
 4   void g() {
 5     for (int m : k) {
 6       switch ((m >>> 1) % 10 + 36)
 7       case 36:
 8         for (int w = -2967; w < 4342; w += 4);
 9         l += 2;
10       case 40: break;
11       case 41: k[1] = 9;
12       }
13   }
14   void o() {
15     if (z) { return; }
16     g();
17   }
18   void p() {
19     for (int q = 2; q < 5; ++q) {
20       z = true;
21       for (int u = 0; u < 9676; u++) o();
22       z = false;
23       o();
24     }
25     System.out.println(l);
26   }
27   public static void main(String[] q) {
28     C t = new C(); t.p(); t.p();
29   }
30 }
```

Fig. 3. JDK-8288975 triggers a mis-compilation in HotSpot. JavaFuzzer generates the seed while `Artemis` inserts the highlighted code. The mutant is reduced by Perses and C-Reduce, and manually cleaned up then simplified for shown in this paper.

The root cause of the issue is that the Expressions Simplification pass incorrectly identifies the invariant of the loops at Lines 5–7. Consequently, the JIT compiler calculates an incorrect number (142 instead of 426) of iterations when hoisting the expression at Line 8 out of the loop at Line 5. To fix this, the developers modified the invariant-check policy in OpenJ9.

It is important to note that this bug manifests only when `C.b()` is JIT-compiled and `C.f()` is hot enough for the inserted loop to be OSR-compiled at the `scorching` level. In contrast, JIT-compiling `C.b()` and scorchingly JIT-compiling `C.f()` of the seed program[2] cannot trigger the bug, as it does not involve OSR. While it is theoretically possible for a test generator to create a program capable of triggering this bug without `Artemis`, the time required for such an attempt is generally unpredictable. However, `Artemis` triggered this bug within eight mutations of the seed program.

HotSpot JDK-8288975. Figure 3 triggers a mis-compilation in HotSpot. This issue was detected by `Artemis` in OpenJDK 11.0.15 (revision f915a327), but it also impacts JDK 17 and 20.

In this example, the seed program invokes `C.g()` only six times (by calling `C.p()` twice at Line 28), which means that no JIT-compilation thresholds are reached. Existing techniques, such as JavaFuzzer, intentionally avoid lengthy loops, as most generated seeds would otherwise require an excessive amount of time to execute. To explore the compilation space, `Artemis` attempts to guide HotSpot to partially JIT-/OSR-compile certain code segments by:

(1) Pre-invoking `C.o()` 9,676 times (Line 21). To preserve semantics, `Artemis` inserts a control flag `z` and a prologue in `C.o()` to facilitate an early return on invocations from Line 21.
(2) Heating up `C.g()` to higher-level JIT compilations by introducing a loop at Line 8.

Despite their simplicity, our mutations trigger non-trivial JIT compilations in HotSpot. The method `C.o()` is first compiled by C1 at the `L3` optimization level, and then further compiled by C2 at the `L4` level. The invocation of `C.o()` at Line 23 leads to a de-optimization because C2 speculatively assumes `z == true` at Line 15. Additionally, the loop at Line 8 is OSR-compiled by C2 at the `L4` level, speculatively assuming that `w` is less than 4,342; it consequently gets de-optimized

[2] This can be accomplished by `-Xjit:{T.b()I}(count=0),{T.f()V}(count=0,optLevel=scorching)`.

when the loop exits. Finally, `C.g()` is also JIT-compiled at the `L4` level. Consequently, HotSpot mis-compiles the mutant, resulting in a different output for `C.l` compared to the seed.

The root cause of this issue is that the Global Code Movement pass incorrectly moves a memory-writing instruction (`storeI`) from an outer loop to an inner loop, as their estimated frequencies are considered equivalent. However, the inner loop actually executes three additional iterations compared to the outer loop. To address this problem, the developers prevented this pass from moving memory-writing instructions into loops that are deeper than their home loops.

SpiderMonkey Bug. Figure 4 illustrates a mis-compilation in SpiderMonkey. `Apollo` identified it in revision f09e3f96.

The seed program defines two functions: Function `f3()` updates `v0` using `a5` based on the value of `a4` (Lines 10–15); Function `f46()` calls `f3()` twice and returns its argument list (Lines 25–28). The seed program finally invokes `f46()` and prints `v0[0]` (Line 29). Notably, no functions are JIT-compiled during their execution in the seed program. `Apollo` alters this scenario similarly to the HotSpot example. Specifically, it ensures that `f3()` gets JIT-compiled by SpiderMonkey's IonMonkey compiler by inserting a loop that calls `f3()` 921 times at Lines 19–23. To prevent `v0` from being updated while `f3()` is called at Line 22, `Apollo` generates an early-return prologue (Lines 4–9) before any statement in `f3()` and controls the prologue's execution by the variable `v2` (Line 2). It is noteworthy that, unlike HotSpot — a statically typed JVM — IonMonkey speculatively assumes that `a4` is of type `double` and `a5` is `null` (Line 22) while compiling `f3()`. However, these assumptions are violated at Line 26, resulting in the de-optimization of `f3()`. Since `Apollo`'s mutations are neutral, SpiderMonkey should yield the same output before and after our mutations. However, it mis-compiled the mutant and produced a different `v0[0]` instead.

```
 1 const v0 = [0xAB0110, {}];
 2 let v2 = false;
 3 function f3(a4, a5) {
 4   if (v2) {
 5     function f9() {}
 6     const v10 = [64,17653,-21120,-2];
 7     v10.unshift(f9, f9, ...v10, ...v10);
 8     return null;
 9   }
10   if (a4 == "g") { v0[0] += a5; return; }
11   let v20 = v0[1][a4];
12   if (!v20) { v20 = 0; }
13   if (v20 < 50) {
14     v0[0] ^= a5; v0[1][a4] = v20 + 1;
15   }
16 }
17 v2 = true;
18 try {
19   for (let i = 0; i < 921; i++) {
20     const v31 = [...]; v31.length >>= v31;
21     const v36 = 2.0 && v31; v36 | 3;
22     f3(v36, null);
23   }
24 } catch(e44) {} finally { v2 = false; }
25 function f46(a47, a48) {
26   f3("s0", 22918); f3("s0", 10489);
27   return arguments;
28 }
29 f46(); print(v0[0]);
```

Fig. 4. The example triggers a mis-compilation in SpiderMonkey. Fuzzilli generates the seed while `Apollo` inserts the highlighted code. The mutant is manually cleaned up and simplified for shown.

3 Compilation Space Exploration

This section provides a rigorous description of compilation space (Section 3.1) and CSX (Section 3.2), along with an explanation of how JoNM operates (Section 3.3), particularly focusing on the implementation details of `Artemis` and `Apollo` (Section 3.4). For simplicity, we demonstrate our formalization in this section around method-JITs, where the unit of JIT compilation is a method. However, our concepts and approaches are also applicable to tracing-JITs, as confirmed by our evaluations (Sections 4 and 5), showing that 80% (68 out of 85) of the reported JVM's JIT-compiler bugs are found in HotSpot's C2 and OpenJ9's JITs. Additionally, we assume that any background JIT compilation capabilities (if they exist) are disabled.

Table 1. The default JIT-trace of the seed program shown in Figure 2 when executed in OpenJ9. `C.m` represents the `main()` method, `C.C` denotes the constructor for class `C`, and `S.p` refers to `System.out.println()`. In OpenJ9, the JIT-compilation level-1 is the `cold` level, while level-2 is the `warm` level.

φ	$=$	$\rightarrow^{1}_{\texttt{C.m}} [T_0] \rightarrow^{1}_{\texttt{C.C}} [T_0]$	*// The first calls to `C.m` and `C.C` are both being interpreted.*
		$\rightarrow^{1}_{\texttt{C.f}} [T_0] \rightarrow^{1}_{\texttt{C.b}} [T_0] \cdots$	*// The first calls to `C.f` and `C.b` are both being interpreted.*
		$\rightarrow^{k}_{\texttt{C.f}} [T_0] \rightarrow^{k}_{\texttt{C.b}} [T_0, T_1, T_2] \cdots$	*// The k-th call to `C.b` enables it to be compiled first at level-1 then at level-2.*
		$\rightarrow^{211}_{\texttt{C.f}} [T_0] \rightarrow^{211}_{\texttt{C.b}} [T_2]$	*// The last call to `C.f` and `C.b`. `C.b` was already JIT/OSR-compiled at k-th call.*
		$\rightarrow^{1}_{\texttt{S.p}} [T_0].$	*// The first and also the last call to `S.p`. It is interpreted.*

3.1 Compilation Space modulo LVM

Profiling counters are essential for JIT compilation. In this paper, we quantify the "hotness" of these counters and their corresponding code constructs — methods or loops — using *temperature*.

Definition 3.1 (Thresholds). To facilitate multi-level JIT compilation, an LVM has N compilation *thresholds* defined as $0 \leq Z_1 \leq Z_2 \leq \cdots \leq Z_N \leq +\infty$. In this paper, we define $Z_0 = 0$ and $Z_{N+1} = +\infty$ to simplify the description. These compilation thresholds divide the values of each profiling counter into $N + 1$ ranges, specifically $[Z_i, Z_{i+1})$ for $0 \leq i \leq N$.

Definition 3.2 (Temperature). After bytecode parsing, an LVM for each method m maintains a set of profiling counters $C_m = \{c_0, c_1, \ldots, c_M\}$ that are updated at runtime. Specifically, c_0 serves as the method invocation counter, while $c_1, c_2, \ldots, c_M$ are control-flow counters for tracking loops' back-edges. A counter c is said to have *temperature* $\tau(c) = T_i$ if and only if

$$c \in [Z_i, Z_{i+1}) \wedge 0 \leq i \leq N$$

where the temperature $\tau(c)$ satisfies a total order, meaning $T_i < T_{i+1}$ always holds for $0 \leq i \leq N - 1$.

A method m's temperature $\tau(m)$ is determined by its hottest counter: $\tau(m) = \max_{c \in C_m} \tau(c)$. A method with temperature T_0 means that it is being interpreted, while a method with temperature $T_{i>0}$ signifies that it is being executed with machine code optimized at the i-th level. For the latter case, a method-JIT implementation implies that the entire method m has been JIT/OSR-compiled, but in a tracing-JIT implementation, it only indicates that the hot loop currently being executed within m has been OSR-compiled, leaving all the remainders in their original states.

The program code can be heated up by executing method calls and loops and it can be cooled down by hitting uncommon traps violating speculative assumptions. In this paper, we refer to such program constructs that influence JIT compilations as *JIT-relevant operations*, or JIT-ops for short. Executing JIT-ops in a method m causes $\tau(m)$ to change over time. We use the *temperature vector* $\vec{\tau}(m, i)$ to represent the trace of these temperature changes during the i-th invocation of m. Essentially, the temperature vector illustrates how an LVM compiles and de-optimizes m upon its i-th call. For example, from the temperature vector

$$\vec{\tau}(m, i) \ = \rightarrow^{i}_{m} [T_0, T_1, T_0]$$

we can infer that: The method m is interpreted when it is initially called for the i-th time; It is subsequently heated up and compiled at level-1 through JIT or OSR compilation; However, it eventually gets de-optimized and is re-interpreted until the method call is completed.

Therefore, a JIT-compilation choice of a program can be represented by a sequence of temperature vectors, which we term *JIT-compilation trace* (JIT-trace) in this paper. A JIT-trace, similar to an annotated method call trace, demonstrates how an LVM executes (interprets or compiles) a program *method call by method call*. It should be noted that every program comes with a default JIT-trace for each LVM; this default trace is generated when the program is executed through the LVM with all JIT-compiler options set to their defaults. Table 1 showcases the default JIT-trace of the seed program presented in Figure 2 when executed in OpenJ9. This trace implies that: (1) The first k calls to `C.b()` allow it to be JIT-compiled at level-2; (2) All subsequent calls to `C.b()` directly execute the compiled code; (3) All other methods are continuously interpreted until `C.main()` exits.

Definition 3.3 (Compilation Space modulo LVM)**.** Given a program P and a language virtual machine LVM, all JIT-traces that LVM can generate while executing the program P construct the compilation space $\mathbb{S}_{\mathsf{LVM}}(P)$ of the program P modulo LVM:

$$\mathbb{S}_{\mathsf{LVM}}(P) = \{\varphi \mid \mathsf{LVM}(P, \varphi) \neq \bot\}$$

where $\mathsf{LVM}(P, \varphi)$ requires LVM to generate the JIT-trace φ first and then returns the program output after running P following the JIT-trace φ. If LVM cannot generate φ, $\mathsf{LVM}(P, \varphi)$ returns $\bot$.

It is noteworthy that an LVM can compile and de-optimize a method at many[3] program points both within the method body and at its boundaries. Consequently, executing a single program P with n method calls can generate $|\mathbb{S}_{\mathsf{LVM}}(P)| = \Omega(2^n)$ possible JIT-traces for LVM. The advantages of such a vast compilation space are twofold:

- Different JIT-traces are likely to trigger distinct JIT optimization paths and passes.
- Running P within LVM following any φ should consistently return the same program output.

These advantages create a considerably strong *test oracle*, presenting a significant opportunity for JIT-compiler validation in modern language virtual machines.

3.2 Compilation Space Exploration

The test oracle tells that an LVM should always yield the same program output regardless of which JIT-trace $\varphi \in \mathbb{S}_{\mathsf{LVM}}(P)$ is generated when executing the program P. Therefore, the JIT compilers in LVM are buggy if we can find two JIT-traces $\varphi_1 \neq \varphi_2 \in \mathbb{S}_{\mathsf{LVM}}(P)$, where

$$\mathsf{LVM}(P, \varphi_1) \neq \mathsf{LVM}(P, \varphi_2)$$

A systematic JIT-compiler validation exhaustively iterates through every legitimate JIT-trace pair in the space to check the output equivalence. We call this *Compilation Space Exploration* (CSX).

Exploration Orders. The compilation space of a program modulo an LVM is structured as a tree, similar to Figure 1. Each node in the tree represents a method call, labeled by its temperature vector, with the root node denoting the bootstrap entry that invokes the main method. An edge between two nodes indicates the caller-callee relationship. It is noteworthy that any path from the root node to a leaf node constitutes a valid JIT-trace within this compilation space. However, unlike Figure 1, each node may have multiple (≥2) child nodes. This is because contrary to the simplistic two-state assumption, a method call is no longer constrained to be interpreted or compiled. Instead, the number of child nodes depends on the size of the temperature vector's value domain.

Starting from a JIT-trace — typically the default JIT-trace — exploring this tree can follow various orders. We propose three groups of exploration orders in this paper:

- *Ordered Walk*. An ordered walk explores the tree in a specified sequence, such as left-to-right or right-to-left. The advantage of this order lies in its completeness, ensuring that no paths (i.e., JIT-traces) are missed during the exploration. However, it requires determining the value domain of the temperature vector for each method call, which poses a challenging problem.

[3] Even though all program points are theoretically valid points for JIT compilation and de-optimization, not all points are permitted for a specific LVM due to considerations such as performance and the complexity of the implementation.

- *Random Walk*. A random walk involves randomly altering the temperature vector of a method call to obtain a new JIT-trace (i.e., jump to a new path in the tree). Although this order is not complete, its simplicity allows for a fast and thorough exploration of the compilation space, when there is a sufficient time budget.
- *Directed Walk*. A directed walk navigates the exploration toward a specific objective. This objective helps select the next paths that may yield interesting results, such as provoking new optimization passes or triggering crashes or mis-compilations. This order enables a more focused exploration, potentially leading to optimal paths more quickly than random or ordered exploration. However, it is not complete, and devising suitable objectives can be challenging.

Possible Realizations. A straightforward and idealized way is modifying the LVM under validation to expose an interface for free control of the dynamic interleaving between interpretation and JIT compilation. However, such modifications require substantial engineering effort as there are many LVM implementations. Additionally, modifying LVMs while aligning with their original internal assumptions is difficult, because: (1) Practical LVM implementations are specifically designed to allow JIT/OSR compilation and de-optimization only at certain program points, which can make some theoretically valid JIT-traces invalid; (2) The space for even a small program is vast and often difficult, if not impossible, to compute due to implicit built-in library method calls (e.g., a simple `print()` call involves dozens of built-in method calls) and the challenge in enumerating the temperature vector's value domain for each method call. Moreover, this realization is evidently LVM-specific and non-portable, making it difficult to validate a wide range of different LVMs.

A practical realization, although not complete, is to fuzz the JIT compiler-related options of an LVM implementation [27]. Nevertheless, this realization entails (1) substantial expertise and manual effort to understand each JIT-compiler option in order to generate valid JIT-traces and (2) a limited exploration capability constrained by the number and effects of JIT-compiler options offered by each LVM. Furthermore, understanding the options of one LVM cannot be readily applied to others, making this realization non-portable. We have experimented with this realization by randomly selecting compilation thresholds using threshold options for each test program, but our one-week effort did not yield any interesting finding. Our experiences in optimizing compilers like GCC and LLVM also tell us that compiler developers are not willing to fix bugs resulting from rarely used options. These factors motivated us to look for a new CSX realization.

3.3 JIT-Op Neutral Mutation

In this paper, we propose a lightweight and LVM-agnostic strategy that approximates CSX at the source-code level with the assistance of JIT-ops, which we refer to as *JIT-Op Neutral Mutation* (JoNM). We leverage the mechanism for profile-guided speculative compilation/optimization (Section 2.1): Modern LVMs profile the program to identify and JIT-compile "hot code", while leaving "uncommon traps" that enable fallback to interpretation in the event of failed speculations. In this context, we can either insert/remove loops or method calls to enable/disable JIT compilation, or manipulate uncommon cold paths to control whether LVMs fall back to the interpretation mode. To effectively utilize CSX's test oracle for cross-validating output equivalence, we should ensure that our modifications to the program do not change its semantics. And eventually, we can explore the compilation space progressively as more mutants are generated.

In particular, JoNM follows a random walk exploration order that we mentioned before. Given a seed program P, JoNM stochastically samples a corpus of methods within P to insert, delete, or modify the JIT-ops (i.e., method calls, loops, and uncommon traps) within P. These mutations derive a set of P's mutants $\mathcal{P}$. JoNM guarantees that the mutations are *neutral* with respect to P's semantics. Specifically, every generated mutant $P' \in \mathcal{P}$ is designed to: (1) Follow a different

Algorithm 1: Validating JIT compilers

```
1 procedure ValidateJITs(LangVM LVM, Program P)
2     R ← LVM(P)                           // We assume R succeeds
3     for i ← 1 . . . MAX_ITER do
4         P′ ← JoNM(P)
5         R′ ← LVM(P′)                     // We execute the mutant
6         if ExecutionFail(R′) then
7             ReportCrash(P′)
8         else if R′ ≠ R then
9             ReportMiscompilation(P′)
```

Algorithm 2: Mutating with JIT-ops

```
1 function JoNM(Program P)
2     P′ ← P
3     foreach Method m ∈ P′.Methods() do
4         if FlipCoin() then   // We follow random walk
5             φ ← Random({LI, SW, MJ, MD})
6             ρ ← Random(m.ProgPoints())
7             L ← SynConstruct(φ, ρ)
8             P′ ← φ.Mutate(P′, m, ρ, L)
9     return P′
```

JIT-trace from P by JIT/OSR-compiling a distinct code segment or by de-optimizing at a distinct program point; (2) Preserve the same program output as P by incorporating for example additional flags and checks. In this way, the thorough exploration of P's compilation space modulo LVM can be approximated by running a sufficient number of mutants of P. Consequently, a JIT-compiler bug exists in LVM if we can find $P' \in \mathcal{P}$ such that

$$\texttt{LVM}(P) \neq \texttt{LVM}(P')$$

Note that we intentionally omit the JIT-trace argument and refer directly to LVM(P) when we require LVM to execute P following the default JIT-trace, in order to simplify the description.

Advantages. Compared to other realizations, JoNM offers several advantages:

- *Lightweight and Simple*: JoNM approximates CSX at the source level, requiring negligible manual effort to understand LVM internals and no modifications to the LVM implementations. Artemis operates directly on Java source code, while Apollo works with FuzzIL [20].
- *LVM-agnostic and Widely Applicable*: Since JIT-ops are usually similar across implementations of the same type of LVMs (e.g., HotSpot and OpenJ9 for JVMs, and V8 and SpiderMonkey for JSEs), a single implementation of JoNM can test various LVM implementations. Artemis and Apollo respectively work for a wide range (≥4) of JVMs and JSEs.
- *Useful and Practical*: JoNM can generate mutants based on real-world programs as well as those produced by program generators. Therefore, (1) any found bug is likely to impact real-world users and vendors and (2) it can empower any given program generator with the capability of JIT-compiler validation. Artemis augments JavaFuzzer while Apollo enhances Fuzzilli.

3.4 The Artemis and Apollo Implementations

We implemented Artemis and Apollo based on JoNM to validate the JIT compilers of JVMs and JSEs. Both tools primarily focus on synthesizing *neutral program constructs* (or construct for short) using two types of JIT-ops: method calls and loops. The utilization of uncommon traps is left to our future work. Following JoNM, the requirement for these synthesized constructs is to preserve the semantics of the program under mutation while capable of introducing a distinct JIT-trace.

Algorithm 1 outlines the main process for JIT-compiler validation. For each seed program P, we mutate it (Line 4) and execute the mutant P' following its default JIT-trace (Line 5) for MAX_ITER times (Line 3). Since the mutations are designed to be neutral, a bug is reported if the execution fails (Lines 6–7) or if there is an output discrepancy between P and one of its mutants (Lines 8–9).

JoNM explores the compilation space in the order of random walk, as shown in Algorithm 2. It operates on the exclusive methods of P', i.e., those defined or overridden within P'. Specifically, it randomly (Line 4) selects a corpus of P''s exclusive methods (Line 3) and then mutates them through four predefined, semantics-preserving mutators (Line 5): *Loop Inserter* (LI), *Statement*

Algorithm 3: Synthesizing program constructs

```
function SynConstruct(Mutator $\phi$, ProgramPoint $\rho$)
    $L \leftarrow \phi$.construct_skeleton    // The construct is initialized as the mutator's construct skeleton
    $V \leftarrow \rho$.Variables()    // The method Variables() returns the set of variable available at the program point $\rho$
    $V' \leftarrow \emptyset$    // This is to save all reused variables during the synthesis process
    foreach ExprHole $\hbar \in L$.expr_holes do
        $L \leftarrow$ Substitute($L$, $\hbar$, SynExpression($\hbar, V, V'$))
    foreach StmtsHole $\hbar \in L$.stmts_holes do
        $L \leftarrow$ Substitute($L$, $\hbar$, SynStatements($\hbar, V, V'$))
    foreach Variable $v \in V'$ do
        $L \leftarrow$ Backup $v$; $L$; Restore $v$;    // We back up $v$'s value before executing $L$ and restore it afterwards
    return $L$    // All left placeholders in $L$ will be subsequently filled by the mutator $\phi$
function SynExpression(ExprHole $\hbar$, VarSet $V$, VarSet $V'$)
    $C$ = GetType($\hbar$)
    if $C$ is a primitive-alike type then
        /* Rule 1: Return a random value with the primitive alike type $C$ within $C$'s domain range. */
        /* Rule 2: Return a random variable $v \in V$ with type $C$; Meanwhile expand $V'$ by $V' \leftarrow \{v\} \cup V'$. */
    else if $C$ is an array type then
        /* Rule: Create an array with dimension $C$.dimen and random size; Let each array element as an
           expression hole typed $C$.ele_type and fill them by SynExpression; Return the array finally. */
    else if $C$ has a non-parameter constructor then return C()
    else return null
function SynStatements(StmtsHole $\hbar$, VarSet $V$, VarSet $V'$)
    $S \leftarrow$ Random(StmtSkeletonDB())    // We collected a corpus of statement skeletons
    foreach ExprHole $\hbar \in S$.expr_holes do
        $S \leftarrow$ Substitute($S$, $\hbar$, SynExpression($\hbar, V, V'$))
    return $S$
```

Wrapper (`SW`), *Method Invocator for JIT* (`MJ`), and *Method Invocator for DeOpt* (`MD`). The mutators operate on an arbitrary program point ρ (Line 6). Their mutation leverages a synthesized program construct L, which may increase or decrease the temperature of method m at program point ρ (Line 7), leading to a new temperature vector and thereby a new JIT-trace. Finally, the synthesized L is inserted into the program at the specified point ρ using the selected mutator ϕ (Line 8).

Construct Synthesis. `SynConstruct` (Algorithm 3) follows the paradigm of programming by sketch to synthesize L; it synthesizes a construct by filling holes left in a predefined skeleton [58, 80]. In this paper, we design three types of holes for a constructed skeleton: *expression holes* (`<expr>`), *statement holes* (`<stmts>`), and *placeholders* (`<placeholder:*>`). Expression holes and statement holes are filled with synthesized Java/JavaScript expressions and statements. These two types of holes typically have minimal connections to any mutators and seed programs. Placeholders, on the other hand, are a special type of expression and statement holes whose semantics are defined by specific mutators. These holes are usually filled directly by Java/JavaScript expressions or statements selected from the seed programs themselves. Figure 5 displays the construct skeletons of the four predefined mutators. Specifically, the skeletons of `LI`, `SW`, and `MJ` are reliant on the loop JIT-op, aiming to enable the mutated method to be JIT-/OSR-compiled. We equip the loop headers in their skeletons with customizable parameters `{MIN,MAX}_{BEG,END,STEP}` to (1) explore different JIT-compilation levels across various JVMs and JSEs and (2) to prevent the loop from running overly long. As for the `MD` mutator, its skeleton is based on the method-call JIT-op. This mutator is designed to de-optimize a method that has already been compiled.

Given a mutator ϕ and a program point ρ, `SynConstruct` synthesizes a neutral construct L by filling ϕ's construct skeleton leveraging the set of variables V available at ρ. It first synthesizes an

```
int b = minmax(MIN_BEG, MAX_BEG, <expr>);
int e = minmax(MIN_END, MAX_END, <expr>);
int s = minmax(MIN_STEP, MAX_STEP, <expr>);
```

(a) The prologue defining a loop's beginning index (b), ending index (e), and iteration step (s). The method `minmax(a, b, x)` returns x if it falls within the a–b range, or a if x is smaller; otherwise, it returns b.

```
for (int i = b; i < e; i += s) {
  <stmts>;
}
```

(b) `LI`'s construct skeleton: `LI` constructs a simple loop from the beginning index b, ending index e, and iteration step s (defined in the prologue). The loop iterates over a synthesized list of statements.

```
boolean exec = false;
for (int i = b; i < e; i += s) {
    <stmts>;
    if (!exec) {
        <placeholder:wstmt>;
        exec = true;
    }
    <stmts>;
}
```

(c) `SW`'s construct skeleton: `SW` wraps a randomly selected statement (which is filled into `<placeholder:wstmt>`) by a loop defined from b, e, and s. In order to preserve the semantics of the seed program, `SW` ensures the selected statement to execute only once through the boolean flag exec.

```
for (int i = b; i < e; i += s) {
    <stmts>;

    P.ctrl = true;
    <placeholder:mcall>;
    P.ctrl = false;

    <stmts>;
}
```

(d) `MJ`'s construct skeleton: `MJ` calls certain method m many times in a loop. It selects a call to m from the seed program P and accordingly synthesizes a new call to m at `<placeholder:mcall>`. To keep semantics, it introduces a boolean flag `ctrl` into P to force m to return early when calling at `<placeholder:mcall>`.

```
if (is_the_last_few_iterations) {
    <stmts>;
    P.ctrl = true;
    <placeholder:mcall>;
    P.ctrl = false;
    <stmts>;
} // the statement will be inserted into in a loop
```

(e) `MD`'s construct skeleton: `MD` calls certain method m at the last few iterations of a loop with totally different arguments by synthesizing a method call expression at `<placeholder:mcall>`. To keep semantics, it introduces a boolean flag `ctrl` into P to force m to return early when calling at `<placeholder:mcall>`.

```
if (P.ctrl) {
  <stmts>;
  return <expr>;
}
// ...
// the original code of the method remains untouched
// ...
```

(f) `MJ` and `MD` insert a prologue instantiated by this skeleton into the method being called. This prologue checks whether `ctrl` is set. Once `ctrl` is set, it causes the method to return early, thereby avoiding the execution of the original method code and preventing any change to its semantics.

Fig. 5. Construct skeletons of `LI`, `SW`, `MJ`, and `MD`. Symbols `<expr>`s and `<stmts>`s are expression and statement holes that should be synthesized when synthesizing constructs, respectively; yet `<placeholder:*>`s are placeholders that should be substituted when the corresponding mutator is making mutations. Hyper-parameters MIN_BEG, MAX_BEG, MIN_END, MAX_END, MIN_STEP, and MAX_STEP are customizable; they work for mutators whose skeletons are reliant on loops. Each variable (e.g., b, exec, `ctrl`) is assigned a unique name to prevent conflicts with all other variables.

expression for each `<expr>` and a statement list for each `<stmts>`, respectively. Then, it substitutes the holes in L with the correspondingly synthesized expressions (Lines 5–6) or statement lists (Lines 7–8). It is important to note that `SynConstruct` does not fill any `<placeholder:*>` holes. Instead, all placeholders are filled when the corresponding mutator is performing its mutations in ϕ.`Mutate`. Additionally, `SynConstruct` backs up the value of every reused variable with a set V' (Line 4) and restores their values afterward (Lines 9–10) to ensure semantics-preserving, as the synthesized expressions or statement lists may modify the reused variables.

» *Expression Synthesis.* `Apollo` synthesizes expressions using Fuzzilli. For `Apollo`, `SynExpression` synthesizes an expression for an `<expr>` hole $\hbar$ based on its type C (Line 12):

- For primitive-like types (Line 14), which include boxed/unboxed [50] primitive types and `String`, `SynExpression` either (1) generates a random value of type C or (2) reuses an existing C-typed variable $v \in V$. In the latter case, it saves the reused variable v to V'.
- For array types (Line 15), `SynExpression` first creates an array instance with the element type C.`ele_type`, the dimension C.`dimen`, and a random size for each dimension. It then treats each array element as an `<expr>` hole of type C.`ele_type` and recursively invokes `SynExpression` to synthesize an expression for each element. Finally, it returns the created array.
- For reference types (Line 16), `SynExpression` always creates a new object if there is a non-parameter constructor. Otherwise, it returns `null` (Line 17). `SynExpression` does not reuse reference variables as accessing their fields/methods may implicitly modify their values.

» *Statement Synthesis.* `Apollo` synthesizes statements from scratch using Fuzzilli. For `Artemis`, we collect a corpus of statement skeletons from the test suites of HotSpot, OpenJ9, and ART, following existing practices in LVM testing [22]. Each statement skeleton consists of a sequence of Java statements containing only `<expr>` holes. `SynStatements` randomly selects a statement skeleton (Line 19) and synthesizes an expression for each expression hole within it (Lines 20–21).

In fact, `<stmts>` and statement skeletons are not a must for JoNM. However, their inclusion significantly diversifies the synthesized construct L in terms of control and data flow. This increased diversity allows L to trigger various optimization paths and passes in the JIT compilers. Additionally, when combined with V', it helps prevent L from being optimized away by the JIT compilers.

Mutators' Mutations. We design four mutators: `LI`, `SW`, `MJ`, and `MD`. They employ different policies for modifying the temperature. Generally, these mutators first finalize the synthesized construct L by filling all `<placeholder:*>`s and then insert L into P'. This process is carried out in ϕ.`Mutate`.

» *Loop Inserter.* `LI.construct_skeleton` (Figure 5b) is a loop that does not contain any `<placeholder:*>`. `LI` directly inserts L into the program point ρ. As a result, the synthesized loop is likely to heat up m to be OSR-compiled at certain compilation levels. Depending on the LVM's speculations, executing this loop may lead to an additional de-optimization when the loop exits.

» *Statement Wrapper.* `SW` (Figure 5c) firstly replaces `<placeholder:wstmt>` with the statement s immediately following ρ, then removes s from P', and finally inserts L at the program point ρ. As a result, the statement s becomes wrapped by the synthesized construct (a loop), significantly changing the control- and data-flow at ρ. To maintain the same semantics, `SW` ensures that the wrapped statement s is executed only once by introducing a control flag `exec` (Line 1). Similar to `LI`, `SW` can trigger OSR compilations (and potentially lead to de-optimizations, depending on the LVM).

Note that the essential difference between `LI` and `SW` becomes apparent when they are applied to tracing-JITs: `SW` causes the wrapped statement and the inserted loop to be compiled together, whereas `LI` exclusively JIT-compiles the inserted loop. Consequently, they induce different control and data flow optimizations within the JIT compiler.

» *Method Invocator for JIT.* In addition to OSR compilation, `MJ` (Figure 5d) is designed to trigger JIT compilation through method calls. Specifically, `MJ` first randomly selects a method call expression k from P' where k directly calls m. It then copies k into k' and synthesizes k''s arguments using `SynExpression` and the following skeleton

```
m(<expr>, <expr>, ...);   // Each <expr> maps one-to-one to the argument in k, preserving the same type.
```

It is important to note that, `MJ` ensures that the number and type of arguments in k' remain the same as those in k during synthesis. Afterward, `MJ` substitutes `<placeholder:mcall>` (Line 5) with k' and inserts the finalized L right before k. Such mutations prompt an LVM to JIT-compile m before k, allowing k to be executed with its machine code after JIT compilation.

However, introducing additional method calls may change the semantics. To mitigate this risk, `MJ` synthesizes another piece of code using the skeleton displayed in Figure 5f which contains a control flag `P.ctrl`. This synthesized code is inserted as the very first statement of m. Within L, `P.ctrl` is set to true (Figure 5d, Line 4) before invoking m (Figure 5d, Line 5) and reset to false afterward (Figure 5d, Line 6). Consequently, running L ensures that no other code in m, except for the synthesized code, is executed, allowing m to consistently early returns.

Figure 3 provides a concrete example for `MJ`. In this example: (1) The method m is `C.o()`; (2) The highlighted code at Lines 20–22 is our synthesized construct L, which pre-invokes `C.o()` for 9,676 times; (3) The method call `o()` at Line 23 is the call we selected. To preserve semantics, a control flag `z` is introduced to class `C` at Line 2. `z` is set to `true` before invoking `C.o()` (Line 20). During the invocations, our synthesized code highlighted at Line 15 is executed and causes an early return, leaving the other statements of `C.o()` unexecuted. Afterward, `C.z` is reset to false (Line 22), allowing our selected call to execute normally. It should be noted that this example is simplified for presentation; the original synthesized code is much more complex.

» *Method Invocator for DeOpt.* `MD` (Figure 5e) operates similarly to `MJ` but specifically focuses on de-optimizing m if it has already been JIT-/OSR-compiled. First, `MD` searches for a method call expression k from P' where k directly calls m and k resides within a loop. It then copies k into k' and synthesizes the arguments of k' using `SynExpression` and the following skeleton

```
m(<expr>, <expr>, ...);   // Each <expr> maps one-to-one to the argument in k, but with a distinct type.
```

Different from `MJ`, `MD` requires that the type of each argument in k' differs from that in k in order to trigger de-optimization. Next, `MJ` substitutes `<placeholder:mcall>` (Line 4) with k' and inserts the finalized L right before k. To maintain semantics, `MD` also introduces a control flag `P.ctrl`, instantiates the prologue skeleton (Figure 5f), and inserts the synthesized prologue before the first statement of m. It is noteworthy that the skeleton of `MD` ensures that k' is executed only in the last few iterations of the residing loop (Line 1); this timing allows k to trigger the JIT compilation of m.

Other Considerations. Although we intentionally preserve the semantics, the performed mutations so far are not entirely neutral. This is because the synthesized code may exhibit unexpected behaviors, such as throwing exceptions. Thus, all four mutators — after their respective mutations — apply the following three final steps: (1) Rename every variable in L with a new name to avoid name conflicts; (2) Disable `stdout` and `stderr` before executing L and re-enable them afterward to avoid unexpected output; (3) Catch and discard any exceptions likely to be thrown by L.

Implementation Details. We have implemented `Artemis` on top of `LI`, `SW`, and `MJ` in ~3,000 lines of Java and ~2,000 lines of Python. It utilizes the Spoon framework [54] for parsing the Java source code and for enabling skeleton definition and instantiation capabilities. Additionally, we extracted a total of 7,823 statement skeletons by parsing the existing test suites of HotSpot, OpenJ9, and ART, following existing practices in LVM testing [22]. We built `Apollo` on top of Fuzzilli [20]. In particular, we implemented a new fuzz engine called `hybrjdon` which interchangeably executes common mutations and JIT-op neutral mutations. This effort includes adding ~1,200 lines of Swift to Fuzzilli. `Apollo` supports all four mutators. We include more details of `Apollo` in Section 5.

4 Validating Java Virtual Machines with 🦌 `Artemis`

The evaluation of our approach aims to validate three widely used production JVMs: HotSpot, OpenJ9, and Android Runtime (ART), using `Artemis`. Highlights of our results on JVMs include:

- **Many detected bugs**: We reported 85 bugs, of which 53 have been confirmed or fixed by the corresponding developers. The reported bugs impact all three validated JVMs.
- **All JIT-compiler bugs**: All reported bugs manifest only when JIT compilers are enabled. These issues do not appear when being executed only in the interpretation mode.
- **Many serious bugs**: Many of the reported bugs are critical, blocking the development of the next release or remaining long-latent across several major releases.

Through such results, we believe that: (1) The quantity and quality of our reported bugs demonstrate the effectiveness of our approach in validating JIT compilers; (2) The discovery of at least 16 bugs per JVM indicates the general applicability of our approach.

4.1 Validation Setup

Selected JVMs. Our evaluation focused on three widely used production JVMs: HotSpot, OpenJ9, and ART. We chose HotSpot and OpenJ9 due to their popularity, following existing research [10, 11, 83]. ART was chosen because of its substantial user base [13]. The open-source nature and their active bug-tracking systems also facilitated us to track bugs, discussions, and fixes. For HotSpot and OpenJ9, we chose to test JDK 8, 11, and 17 because they are long-term supported (LTS). ART was excluded from this selection since it does not directly support class bytecode.[4] For each selected JVM, we built its latest trunk and validated it with (1) background compilation (if supported) disabled and (2) a Java heap memory allocation of 1 GiB. We opted not to use the latest stable releases, as their bug fixes are only reflected in subsequent stable versions. This time gap would hinder us from distinguishing whether a newly detected bug duplicates an existing one. Finally, our evaluation primarily focused on the x86_64 Linux platform.

Seed Programs. We used JavaFuzzer [21], a random Java program generator, to create seed programs for `Artemis`, as we observed that JavaFuzzer-generated programs are typically complex and offer rich opportunities for `Artemis`' mutation. Additionally, our experience using JavaFuzzer tells us that the Java programs it created can be effectively reduced with a combination of Perses [60] and C-Reduce [56]; this minimizes our efforts in reporting bugs to the developers. However, it is important to note that `Artemis` is agnostic to seeds, meaning that it can be integrated with other Java program generators or even real-world programs. We did not use them mainly because the reduction of tests generated by such programs often takes a considerable amount of time.

Synthesis Parameters. Our experience with hundreds of attempts suggests that using eight mutants strikes a good cost/effectiveness balance for exploring the compilation spaces of the seed programs generated by JavaFuzzer. Therefore, in our evaluation, we set `MAX_ITER` to 8 to approximate the exploration of eight JIT-traces for each seed program. Since different JVMs define varying default compilation thresholds, we adjusted `{MIN_BEG,MAX_BEG}–{MIN_END,MAX_END}` accordingly: `{-100,+100}–{4900,5100}` and `{-100,+100}–{9900,10100}` for HotSpot/OpenJ9 while setting them to `{-100,+100}–{19900,20100}` and `{-100,+100}–{49900,50100}` for ART. We set `MIN_STEP–MAX_STEP` to `0–10` for all of them.

4.2 Quantitative Results

Numbers of Bugs. We have filed a total of 85 bugs across the three JVMs, including 32 in HotSpot, 37 in OpenJ9, and 16 in ART. Among them, 17 bugs were found in method-JITs (i.e., HotSpot's

[4] ART natively supports dex bytecode which are transpiled from class bytecode.

Table 2. The statistics of reported JIT-compiler bugs and the affected JIT-compiler components due to crashes. The columns "#" indicate the number of JIT-compiler crashes affecting the corresponding component. "Code Execution" represents that a crash occur while the JVM is executing the compiled machine code. "Other JIT Components" includes modules handling JIT-INT interaction, synchronization, etc. "Garbage Collection" refers to instances where the JIT compiler triggers a crash in the garbage collector.

(a) Reported JIT-Compiler Bugs

	HotSpot	OpenJ9	ART	Total
Reported	32	37	16	85
Numbers of reported JIT-compiler bugs				
Duplicate	8	5	2	15
Confirmed	22	19	12	53
Fixed	4	12	10	26
Types of reported JIT-compiler bugs				
Mis-comp.	1	9	8	18
Crash	30	28	8	66
Performance	1	0	0	1

(b) Affected JIT-Compiler Components (Crashes)

HotSpot Component	#	OpenJ9 Component	#
Inlining, C1	1	Local Value Propa.	1
Ideal Graph Building, C2	4	Global Value Propa.	2
Ideal Loop Optimizat., C2	10	Loop Vectorization	1
Global Constant Prop., C2	1	De-optimization	1
Global Value Number., C2	5	Register Allocation	1
Escape Analysis, C2	1	Code Generation	2
Register Allocation, C2	2	Recompilation [48]	1
Code Generation, C2	3	Other JIT Compone.	6
Code Execution, C2	3	Garbage Collection	13

C1 and ART's OptimizingCompiler) and 68 bugs in tracing-JITs (i.e., HotSpot's C2 and OpenJ9's JITs). The first half of Table 2a summarizes their status, where 53 have already been confirmed and 26 have been fixed. We recognize a reported bug as "Confirmed" if the corresponding JVM developers can reproduce it in their environment. Otherwise, we leave the bug in the "Reported" category regardless of whether or not we have a complete crash log for reproduction and diagnosis. While we ensured that all reported bugs exhibit different symptoms (e.g., stack traces), two bugs in ART and five in OpenJ9 still stem from the same root causes as some bugs that we had reported previously. Additionally, we identified eight unique HotSpot bugs duplicating those reported by other developers or users, demonstrating that Artemis can uncover bugs that common users actually encounter in their development. We categorize all these instances as "Duplicate".

Types of Bugs. The reported JIT-compiler bugs can be categorized into the following types:

Mis-compilation. This occurs when the JIT compiler incorrectly compiles the program, leading to a semantic discrepancy between the bytecode and the compiled machine code, meaning that running them produces different outputs. This issue is likely due to (1) bytecode compilation, (2) upper-level optimization, or (3) de-optimization.

Crash. This type refers to instances where JVM crashes either during the compilation of some code or while executing the compiled machine code. Symptoms of such bugs can vary and include segmentation faults, assertion failures, etc.

Performance Issue. In this case, executing the compiled machine code results in the JVM performing significantly slower than interpreting the bytecode. This slowdown is typically noticeable to users and may eventually lead to the JVM process being terminated by the operating system.

The second half of Table 2a categorizes our reported bugs into these three types. More than 20% of the bugs are mis-compilations, i.e., the most interesting and hard-to-detect issues [3]. It is important to note that, prior to our work, we were unaware of any work having the ability to enable a program generator such as JavaFuzzer to find mis-compilations without JVM-oriented differential testing (for example, differential testing of different JVM implementations or of different options of the same implementation). Although we identified many mis-compilations in OpenJ9 and ART, HotSpot remains an exception. This is likely due to HotSpot, as the most prevalent implementation, is much more mature than the others. We found only one performance issue where the HotSpot process running the test was killed on Ubuntu while it operated noticeably slower on Windows.

Importance of Bugs. It is worth mentioning that all the reported bugs are JIT-compiler bugs that would otherwise remain hidden by the bytecode interpreter if JIT compilation was disabled.

In addition, many reported bugs are considered serious. Specifically, 12 out of the 37 OpenJ9 bugs were classified as `blocker` — the most severe, release-blocking type of bugs. We also found 10 out of the 32 HotSpot bugs that were marked as at least `P3` — major loss of function. There have been 13 long-latent OpenJ9 bugs that have persisted across ≥4 major and many minor releases, escaping the testing campaigns by earlier and contemporary tools. The developers expressed surprise at the effectiveness of our tool and even inquired, "*Do you think there are going to be many more?*"

Furthermore, we received very positive feedback from the respective JVM developers:

- HotSpot developers are looking forward to our research: *"I'm *** from the HotSpot Compiler Team at Oracle and I noticed that you filed quite a few bug reports for the JITs recently, thanks a lot for that! ...Is there anything you could share with us? ...I'm looking forward to learning more about your research ..."*
- OpenJ9 developers even invited us to make further contributions with friendly support: *"I'm not sure how you are finding these problems. ...@*** is interested in having you open a Pull Request to deliver the test cases ...We'd try to make it easy so you don't need to be concerned much about test frameworks ..."*

Affected Components. The bugs that we have reported are diverse, affecting various JIT-compiler components, as illustrated in Table 2b. Due to the challenges in identifying affected components for mis-compilations and performance issues (if they are not yet fixed), we focused our analysis on crashes. Additionally, we excluded JVMs with ≤10 crashes, as their results are unreliable.

» *HotSpot.* The 30 crashes affect 8 C1/C2 components, with the majority related to C2. This is reasonable, as C2 is considered significantly more complex than C1, involving more aggressive optimizations. Of the 32 crashes, 29 occur during the compilation of C1 or C2, while the remaining three (i.e., column "Code Execution") occur while executing the compiled code. The most affected component is ideal loop optimization, followed by global value numbering and ideal graph building.

» *OpenJ9.* The affected components in OpenJ9 differ from those in HotSpot. Specifically, the 28 crashes impact ≥8 JIT-compiler components, where 26 occur during the compilation process of OpenJ9's JITs and the other two during code execution. To our surprise, most crashes occur within the garbage collector. After discussing these crashes with OpenJ9 developers, we learned that they are indeed JIT-compiler bugs, as the JIT compiler corrupts heap memory, leading to crashes in the garbage collector. Furthermore, if these heap memory corruptions are mishandled, they could result in serious, exploitable security vulnerabilities [14], indicating that JIT-compiler bugs pose a significant risk as they can affect JVM components beyond the JIT compiler itself [43]. The most affected components within the JIT compiler are global value propagation and code generation.

4.3 Comparative Study

To further investigate the effectiveness of our approach, we conducted a comparative study with the traditional approach to demonstrate the advantages of exploring more JIT-traces.

In this study, we used the synthesis parameters detailed in Section 4.1 and selected OpenJ9 (JDK 11, revision 4ca209b5) as the target for testing. We employed JavaFuzzer as the seed generator. For each seed, we first executed it once with its default JIT-trace. Next, we ran it again by forcing every method to be JIT-compiled before their first invocations by using the `-Xjit:count=0` option, regarding all JIT compilers as static compilers, similar to traditional approaches [29, 67]. We then

Table 3. The results of the comparative study between CSX and the traditional approach and the mutation cost of `Artemis` in seconds. In the left table: The first two columns read the number of seeds and mutants generated; The columns "#CSX" and "#TRAD" list the number of seeds for which the corresponding approach was able to trigger output discrepancies; "#Shared" is the number of seeds that both approach triggered discrepancies. As for the right table: The row "Cost@1" refers to the cost of generating a single mutant via `Artemis`; The row "Cost@n" is the cost when `Artemis` is booted only once but generates multiple mutants.

(a) Results of the Comparative Study

#Seeds	#Mutants	#CSX	#TRAD
42,559	340,472	154	21
—		**#Shared**: 16	

(b) Mutation Cost (in Seconds)

	Mean	Median	Min	Max
Cost@1	1.65	1.68	0.76	2.01
Cost@n	0.16	0.16	0.06	2.19

mutated the seed eight times using `Artemis` and executed each mutant with its default JIT-trace. Finally, we compared their outputs and counted the number of seed programs that resulted in output discrepancies. An approach is considered more effective if the number is greater.

We conducted the study on an AMD server with a Ryzen Threadripper 3990X 64-core processor for a period of seven days. To demonstrate that `Artemis` performs effectively even on commodity machines, we enabled 16 of the available 64 cores. During this process, we discarded seed programs or mutants that did not finish within two minutes.

Results. Table 3a presents the results. Over the 7 days, `Artemis` prompted JavaFuzzer to generate 42,559 seeds and performed 340,472 mutations. Among these seeds, `Artemis` successfully guided 154 to trigger discrepancies, with 89.6% (138) not being identifiable simply by comparing the default and fully compiled JIT-trace, i.e., the traditional approach. There are five seeds for which `Artemis` was unable to trigger any difference within the eight mutants while the traditional approach could. Upon detailed inspection, we found that these seeds involve JIT/OSR compilations of built-in methods that are beyond `Artemis`' current capability. We will discuss this further in Section 6.

4.4 Performance Overhead

We also measured `Artemis`' performance concurrently with the comparative study.

Throughput. Throughput measures the number of LVM invocations per second. During the study process, `Artemis` invoked OpenJ9 at least 383,031 times, achieving a throughput of ≥0.63 OpenJ9 invocations per second. This indicates that `Artemis` can test a program in ~15s (including 9 source-to-bytecode compilations and 10 OpenJ9 invocations). Most of the CPU time is consumed on source-to-bytecode compilation and executing the synthesized loops. Considering that (1) `Artemis` primarily focuses on loops (`LI`, `SW`, and `MJ`) at present, which often prolong the process (typically taking dozens of seconds to finish) and (2) we only enabled 16 cores during the evaluation, we believe that this is a practical throughput.

Mutation Cost. Given a seed program, the cost for `Artemis` to generate a single mutant is low. On average, it took ~1.65 seconds to complete both (syntax and semantic) source parsing and construct synthesis, with the former taking ~0.67 seconds and the latter ~0.88 seconds. In a practical JIT-compiler validation process or JVM fuzzing process, where `Artemis` and its dependent skeleton engine Spoon [54] are booted only once but derive numerous mutants for a variety of seed programs, the mutation cost becomes negligible: It took only ~157 milliseconds to mutate a seed program on average. Table 3b presents more statistics, in which the relatively larger cost of "2.19" occurs only at the very first mutation (when `Artemis` is booted).

```java
class C {
 boolean b;

 void a() {
  int c, v;
  double d = 2.61331;
  for (int z=830; z>51; --z)
    b = b;
  v = 1;
  while (++v < 908)
   for (c=-3230; c<9840; c+=2)
    try {
     MethodHandle m = null;
     m.invokeExact();
    } catch (Throwable t) {
    } finally {}
   System.out.println(d);
 }

 public static void main(...) {
  C t = new C();
  for (;;)  t.a();
 }
}
```

(a) JDK-8287223: Assert. Failure

```java
class C {
 void l() {
  int m, d;
  byte[] e = new byte[6];
  for (int j=2; j<222; ++j) {
   for (m=d=1; d<4; d+=2)
    e[m] -= d;
   for (int z=5; z<948; z++) {
    boolean[] x = new boolean[576];
   }
  }
  System.out.println(f(e));
 }
 long f(byte[] a) {
  long k = 0;  int i = 0;
  while (i < a.length) {
   k += a[i]; i++;
  }
  return k;
 }
 public static void main(...) {
  C t = new C();  t.l();
 }
}
```

(b) Issue-15369: Mis-comp.

```java
class C {
 void e() {
  for (int i=-4605; i<2; i++) {
   try {
    double[] d = new double[1];
    d[0] = 0;
   } catch (Throwable x) {
   }
  }
 }

 public static void main(...) {
  C t = new C();
  for (;;)
    t.e();
 }
}
```

(c) Issue-15305: Segment. Fault

```java
class C {
 long c; int x;
 void f() {
  int i, p, q = 42252;
  int j, g, k = 5;
  double m;
  long[] l = new long[256];
  for (int o=2; o<87;) {
   boolean z = false;
   for (j=737; j<16822; j+=3)
    if (!z) {
     z = true;
     for (p=1; p<6; ++p) {
      for (m=1; m<2; m++) {
       l[(int)m]-=16;  c>>>=o;
      }
      for (g=1; 2>g; g+=3) {
       l[g] -= k; i = (int) c;
       try { k = q / i; }
       catch (ArithExcp w) {}
      }
      switch (o) {
       case 73: x = p;
      }
     }
    }
  }
 }
 void h() { f(); }
 public static void main(...) {
  C t = new C();  t.h();
 }
}
```

(d) JDK-8288190: Fatal Arith. Error

```java
class C {
 long i;

 void p(long l) {
  int f = 0;
  int g[] = new int[0];
  long[] h = new long[0];
  for (int j=7; j<84;)
   try {
    long[] a = {1};
    for (int z=1; z<=f; z++)
     l += a[z - 1];
   } catch (Throwable r) {
   } finally {
    l = 7;
   }
  i = l + y(g) + w(h);
 }

 void k() { p(0l); }

 long w(long[] a) {
  return 0;
 }

 long y(int[] a) {
  return 0;
 }

 public static void main(...) {
  C t = new C();  t.k();
 }
}
```

(e) Issue-15335: Segment. Fault

```java
class C {
 int r;

 void f() {
  int i, j, o = 5788, t = 127;
  byte[] a = new byte[400];
  for (i=14; 297>i; ++i)
   for (j=151430; j<235417; j+=2);
  try {
   for (int d=4; 179>d; ++d) {
    o *= o;
    for (int k=1; k<58; k++)
     for (int z=k; 1+400>z;
        z++) {
       a[z] -= t;
       o += z;
       switch (d % 5) {
        case 107: d >>= r;
       }
      }
   }
  } catch (AIOOBExcp x) {}
  System.out.println(i + "," + o);
 }

 public static void main(...) {
  C t = new C();
  t.f();
 }
}
```

(f) Issue-227365246: Mis-comp.

Fig. 6. A small selection of example tests finding various JVM bugs. For simplicity, we omit the required imports and the parameter of `C.main(String[] args)` method. `ArithExcp` represents `ArithmeticException`. `AIOOBExcp` stands for `ArrayIndexOutOfBoundsException`. Code shown in these examples are cleaned-up versions from very large test programs generated with a combination of JavaFuzzer and `Artemis`.

4.5 More Examples

Artemis has proven effective at identifying a wide range of bugs such as segmentation faults (SIGSEGV), fatal arithmetic error (SIGFPE), emergency abort (SIGABRT), assertion failures, mis-compilations, and performance issues. We discuss a small selection of these findings to highlight their diversity. All tests presented in this section are derived from JavaFuzzer [21] and have been mutated by Artemis. These tests have been reduced from much larger mutants with a combination of Perses, C-Reduce, and further manual cleanup if needed.

Figure 6a. When HotSpot's C1 compiler compiles the given code, it attempts to inline the method `invokeExact()`, which necessitates that the receiver be non-null (Line 14). However, the HotSpot developers overlooked the null check, resulting in an assertion failure.

Figure 6b. The DLT (Dynamic Loop Transfer, which facilitates on-stack replacement) optimization pass in OpenJ9 fails to preserve the type information of the byte array `e` (Line 7), resulting in an unexpected byte-bit truncation.

Figure 6c. OpenJ9 crashes with a segmentation fault when the method `C.e()` is compiled at the `scorching` level. This is because OpenJ9's JIT compiler accesses a removed or invalid block when traversing predecessor blocks to compute and extend the live ranges of some variables.

Figure 6d. When compiling the loop in `C.f()`, HotSpot's C2 compiler produces incorrect code, leading to a crash with a fatal arithmetic error during the execution of the compiled code.

Figure 6e. OpenJ9 fails to vectorize the loop in `C.p()` and as a result, crashes as a segmentation fault because of a null pointer dereference when checking for loop independence.

Figure 6f. When array access is out of bounds in the compiled code, ART is expected to de-optimize along a bounds-check slow path where the exception should be caught if an exception handler is present. However, ART's bounds-check slow path generator fails to compute the correct array index and length, resulting in the exception being caught at an incorrect index.

Figure 7. This represents the only performance issue identified by Artemis. When executing the code OSR-compiled by HotSpot's C2 compiler, the native memory usage increases unexpectedly. This eventually exhausts the machine's memory, leading to the process running the test being terminated by the underlying Ubuntu system (with 16 GiB of RAM). In contrast, while the process does not get killed on Windows (also with 16 GiB of RAM), it becomes significantly slower, even slower than interpreting the bytecode. This issue was confirmed as a potential memory leak bug shortly after we reported it, but after several weeks, it was marked as "`Won't Fix`". The developers hypothesized that the application was simply allocating memory faster than the garbage collector could reclaim it. However, we believe this should be considered a real bug affecting end users because (1) the garbage collector should be opaque to end users and (2) end users should not have to worry about the garbage collector's speed when developing Java applications.

5 Fuzzing JavaScript Engines with 🦅 Apollo

Building on our promising results in validating JVMs, we experimentally implemented our approach specifically JoNM into Fuzzilli [20], a state-of-the-art JavaScript fuzzer with a special focus on crashing bugs. Our goal is to investigate: *Whether JoNM can augment Fuzzilli with the capability to identify mis-compilations without significantly compromising its ability to find crashes.* The outcome is Apollo, an extension of Fuzzilli featuring a new fuzz engine called hybrjdon based on JoNM.

5.1 Apollo and Fuzzilli

Fuzzilli is a hybrid fuzzer that employs both generation and mutation strategies. It is booted up as a generative fuzzer, continuously generating seed programs until no coverage increments are

```java
class C {
 public static void main(String[] g) {
  for (int i=0; i<10; i+=1)
   for (int j=1; j<197; j+=1)
    for (int k=-4910; k<314; k+=1)
     try {
      boolean b = false;
      byte[] a = new byte[1 << 14];
      try (
       ByteArrayOutputStream o = new ByteArrayOutputStream();
       ZipOutputStream z = new ZipOutputStream(o)
      ) {
       final byte[] x = { 'x' };
       z.putNextEntry(new ZipEntry("a.gz"));
       GZIPOutputStream g = new GZIPOutputStream(z);
       g.write(x);  g.finish();
       if (b) z.write(x);
       z.closeEntry();
       z.putNextEntry(new ZipEntry("b.gz"));
       GZIPOutputStream k = new GZIPOutputStream(z);
       k.write(x);  k.finish();
       z.closeEntry();  z.flush();
       a = o.toByteArray();
      }
     } catch (Throwable x) {}
 }
}
```

Fig. 7. JDK-8290360: Performance Issue. When executing the C2 compiled code, the HotSpot process running this test is directly killed on Ubuntu while becomes much slower on Windows. Code shown in this example is a cleaned-up version from a very large test program.

observed over a specified number (100 by default) of consecutive generations. After this boot-up phase, Fuzzilli transfers to a mutative fuzzer, which modifies existing seed programs guided by code coverage. Its default mutation policy, known as the `mutation` fuzz engine, performs five consecutive mutations for each sampled seed program. For each of the five generated mutants, Fuzzilli saves it into the seed corpus for further mutations if its execution results in new code coverage; additionally, if it leads to a new crash, that crash is reported.

`Apollo` differs from Fuzzilli in its JoNM-based fuzzing engine `hybrjdon`. First, since Fuzzilli-generated programs do not produce output, `hybrjdon` inserts a checksum variable along with update operations (such as addition, subtraction, multiplication, etc.) on that checksum variable before applying any JoNM mutators. `hybrjdon` prints the final value of the checksum variable to `stdout` at the end of a sample's execution as its program output. Additionally, rather than performing random mutations, `hybrjdon`'s mutations are limited to `LI`, `SW`, `MJ`, and `MD`. For each generated mutant, `hybrjdon` checks whether the value of the inserted checksum variable is identical to that of the seed program. It reports a mis-compilation for any checksum that differs, or a crash if the mutant leads to a crash. Finally, `Apollo` periodically reverts to the generative and the default `mutation` engine to enhance the diversity of the seed corpus. In the rest of the paper, we will use `Apollo` and `hybrjdon`, fuzzer and fuzz engine interchangeably when the context is clear.

Implementation Challenges. Although conceptually general, implementing JoNM for JavaScript is not as straightforward as for Java. This is because JavaScript is a programming language with dynamic types and treats functions as a first-class member. These features make it more frequent to

Table 4. The list of problems that we encountered during the implementation of `Apollo` and the primary strategies that we employed to try addressing them. In fact, all applied strategies are conservative and may introduce false positives. On the other hand, some of these strategies already exist in Fuzzilli; we reused and extended them for `Apollo`'s specific requirements. The examples shown are kept minimal, with keywords like **`function`** and **`let`** being omitted when defining a function or variable due to space limitations.

Problems	Examples	Strategies
infinite recursion	`f(t){t(t)}; z=f; f(z);`	type tracing, def-use analysis, disallowed explicit recursion
implicit typecasting	`f(){}; for(t of 1+f){};`	type tracing, def-use analysis, disallowed explicit typecasting
prototype pollution	`t.__proto__ = f(){};`	type tracing, limited built-in (e.g., __proto__) access
checksum overflow	`chk = chk / 0 + 1;`	context & scope analysis, bounded checksum

produce programs with indeterministic behaviors resulting from infinite recursion, implicit typecasting (or type coercion), prototype pollution, and checksum overflow among others, potentially breaking the the semantics-preserving requirement of our mutators. To prevent such issues, `Apollo` incorporates additional strategies, some of which are outlined in Table 4. We provide detailed explanations for some of the key strategies below:

» *Type Tracing.* `Apollo` traces the type of each generated code entity, such as variables and functions, and for object literals and their properties, it traces their shapes. This strategy forms the basis for identifying function variables and disallowing them from being used as function or method arguments, thereby potentially reducing the risk of implicit infinite recursion. Recognizing types also helps to eliminate potential implicit typecasting where we expect expression operands to be of the same type, removing implicit `toString` and `valueOf` calls. Additionally, type tracing is particularly beneficial for `MJ` and `MD` mutators, where `Apollo` is required to synthesize arguments with the same or distinct types to enable JIT compilation or de-optimization, respectively.

» *Static Analysis.* `Apollo` analyzes the context and scopes of each program point, def-use relations of variables, and dead code among others. For example:

- Maintaining context (e.g., inside a switch-case or a loop) and scope (e.g., the global scope, function scope, and class method scope) is crucial for all our mutators because they need to work inside a method, and the `MD` mutator requires a loop. Context and scope analysis also helps to track all available variables at a program point. However, unlike in Java, context and scope analysis for JavaScript also requires consideration of object literals, closures, and code strings consumed by `eval()`; improper handling could lead to unexpected insertion of checksum update operations, potentially leading to checksum overflow.
- Through def-use analysis, we can identify existing function calls for `MJ` and `MD`. Additionally, we can recognize references to methods such as `toString()` and disallow unexpected calls, as the method by default returns the function's code if it is not implemented.
- We leveraged dead code analysis to avoid inserting code into the dead code area.

» *Bounded Checksum.* `Apollo` enforces several constraints. Firstly, we restrict the value of the operands that update the checksum variable to be in the bounded range of `[1,25536]`. Secondly, we confine the operators to be one of `ADD`, `MINUS`, `MULTIPLY`, `BIT_ADD`, `BIT_OR`, `BIT_XOR`, `LOGIC_ADD`, `LOGIC_OR`, `LSHIFT`, and `RSHIFT`. We exclude the `DIVISION` and `MODULO` operators, as they introduce `Infinity` when divided by zero. Finally, we limit the total number of checksum update operations by associating each method and function (whether defined globally or locally) with an update count. The checksum variable is updated only when its updates within a function or method do not exceed a predefined threshold, which we set at `50`.

» *Minor Strategies.* Apollo encompasses several minor strategies; these strategies disable many advanced Fuzzilli features. For instance, it prohibits the generation of explicit recursion, explicit calling into certain built-in methods such as toString and valueOf, and explicit access into certain built-in properties such as __proto__. While these constraints are not designed for discovering crashes, they are likely to introduce inaccuracies for mis-compilations; it should be noted that Apollo only enables them for the latter case.

» *Fallback Strategy.* All previous strategies are conservative and may introduce false positives. To mitigate this, Apollo pre-executes a sample three times and only accepts it for our mutation if the output remains consistent throughout all pre-executions.

5.2 Research Questions

To assess whether Apollo can fulfill the aforementioned goal, we experimentally evaluated it on seven popular production JSEs, including three major JSEs, namely V8, JavaScriptCore (JSC), and SpiderMonkey (SM), as well as four embeddable JSEs, namely QuickJS (QJS), JerryScript (JRY), Duktape (DUK), and Nginx JavaScript (NJS). This evaluation is in line with our goal, currently focusing on Apollo's bug-finding capabilities, specifically in terms of code coverage and the number of discovered bugs (crashes and mis-compilations) identified within a limited timeframe. We did not conduct a field fuzzing campaign due to time and resource constraints. We explored:

RQ1 Does Apollo compromise Fuzzilli's ability to achieve code coverage and detect crashes?
RQ2 Does Apollo introduce additional overhead? What is the throughput and execution cost?
RQ3 Can Apollo identify real-world JIT-compiler mis-compilations?

5.3 Evaluation Setup

Selected JSEs. Our evaluation of Apollo focused on the latest trunks of the seven popular engines: V8, JavaScriptCore, SpiderMonkey, QuickJS, JerryScript, Duktape, and Nginx JavaScript. These engines were selected based on their popularity, as noted in existing research [3, 20, 67]. We chose ECMAScript 2015 (also known as ES 6) as our test target due to its widespread adoption and relatively[5] good support across all engines. Our evaluation was performed on the same Linux platform (x86_64) as in our JVM validation experiment (Section 4).

Baseline Engines. In addition to Fuzzilli's default mutation engine and Apollo's hybrjdon engine, we developed a further engine called jitmut, serving as an intermediate baseline between them. The jitmut engine performs random mutations related to JIT compilation, de-optimization, and re-compilation through copying the idea of LI, SW, MJ, and MD, but it does not perform all semantics-preserving operations and does not check the equivalence of program outputs. Its primary focus is on identifying crashes within the JIT compilers.

Answering RQ1. We ran all three fuzzers (i.e., fuzz engines) on the latest trunks of all seven JSEs. For each fuzzer, we deployed two parallel instances, each assigned to 24 separate CPUs, to run for 24 hours. This setup resulted in fuzzing in a total of

$$2 \text{ (instances)} \times 7 \text{ (engines)} \times 3 \text{ (fuzzers)} \times 24 \text{ (CPUs)} = 1,008$$

CPU days (or 24,192 CPU hours). We specially focused on branch coverage. We identified a unique crash by its error message (assertion or stack backtrace). For crashes that we were unable to obtain any error message, we treated each of them as a unique one. It is important to note that we intentionally excluded out-of-memory issues from consideration as most such crashes can be

[5] It is important to note that not all ES6 features are supported by every JSE evaluated. For example, Duktape does not support template strings [15]. We excluded issues (crashes or mis-compilations) arising from unsupported features.

Table 5. The detailed statistics of the 24-hour fuzzing. Symbols within parentheses are units: "%" for percentage, "m" for million, "k" for thousand, "s" for seconds. "Correct." represents the ratio of syntax-valid samples. "IntSamps" measures the number of generated interesting samples that contribute to the increase of code coverage. "CovEffi." indicates the number of interesting samples required to lead to a 1% increase in code coverage. "Execut." is the number of JSE invocations during the 24-hour fuzzing and "ExCost" is the time required to execute a sample. "Thrput" is the number of executions per second.

Metric/JSE	**Major JSEs**									**Embeddable JSEs**												
	V8			**JavaScriptCore**			**SpiderMonkey**			**QuickJS**			**JerryScript**			**Duktape**			**Nginx JavaScript**		
	`mut`	`jit`	`hyb`	`mut`	`jit`	`hyb`	`mut`	`jit`	`hyb`	`mut`	`jit`	`hyb`	`mut`	`jit`	`hyb`	`mut`	`jit`	`hyb`	`mut`	`jit`	`hyb`
#Crashes	0	0	0	0	0	0	0	0	0	105	2,252	41	2,579	2,681	2,345	207	180	160	182	167	151
General Statistics																					
Coverage (%)	12.24	12.14	11.21	19.52	20.07	17.51	23.01	23.77	21.35	45.27	46.62	46.7	68.21	68.1	67.3	36.84	36.79	37.33	48.76	49.23	47.66
IntSamps (k)	3.96	3.2	**2.36**	4.9	5.1	**3.12**	6.32	6.87	**5.29**	4.04	4.24	**4.01**	**2.96**	2.97	3.08	2.16	**2.12**	2.13	2.68	2.73	**2.56**
CovEffi. (#/%)	323	264	**211**	251	254	**178**	274	289	**248**	89	91	**86**	**43**	44	46	59	58	**57**	55	55	**54**
Correct. (%)	66.75	67.51	**74.15**	66.61	65.93	**74.61**	68.14	67.22	**76.48**	19.94	18.05	**77.86**	51.91	49.85	**67.27**	14.7	12.89	**29.17**	11.07	9.68	**19.2**
Performance Statistics																					
Execut. (m)	2.34	1.8	1.78	2.59	2.97	2.64	2.94	3.5	3.73	34.68	42.58	10.63	12.7	13.02	21.73	27.15	24.02	28.86	78.66	94.56	111.68
ExCost (s)	2.15	1.67	1.97	2.11	1.69	2.15	0.57	0.59	0.43	0.3	0.05	0.04	0.02	0.04	0.04	0.18	0.21	0.05	0.12	0.14	0.07
Throput (#/s)	18.55	17.83	25.16	11.28	20.53	20.53	33.56	36.27	29.42	225.45	534.16	169.89	49.81	55.76	164.82	348.2	310.03	401.94	896.82	820.48	880.53

mitigated by allowing a larger heap memory. We plotted and analyzed the average data of the two instances for each combination of a fuzz engine and a JSE.

Answering RQ2. We collected the running stats during the 24 hours and assessed the performance overhead of Apollo based on: (1) Throughput, which measures the number of JSE invocations per second; (2) Execution cost, which indicates the time required to execute a sample.

Answering RQ3. We ran Apollo for one week on the latest trunks of the three major JSEs — V8, JavaScriptCore, and SpiderMonkey — to find out whether Apollo is capable of finding real-world mis-compilations. Specifically, for each JSE, we deployed one Apollo instance with 64 separate CPUs and 32 GiB RAMs. We collected all reported mis-compilations for analyses.

Synthesis Parameters. To maintain a fair comparison, we set `MAX_ITER` to `5` to enable five consecutive mutations following Fuzzilli's default settings. As for `{MIN,MAX}_{BEG,END}`, we set them to a fixed number `921` to only allow a fixed number of loop iterations, as the default Fuzzilli tunes down the compilation thresholds of all JIT-compilers. We set `{MIN,MAX}_STEP` to `1`.

5.4 RQ1. Coverage and Crashes

We present the overall statistics from the 24-hour fuzzing in Table 5. In this research question, we focus specifically on the "General Statistics" section. Figure 8 presents the coverage trend during the 24-hour fuzz campaign for each fuzz engine and JSE.

Overall, despite disabling many advanced features that may introduce indeterministic behaviors but significantly contribute to code coverage, the `hybrjdon` engine still achieved comparable results. In particular, the coverage for all three engines was similar, with only a marginal difference of less than 2.55%. The coverage for the four embeddable JSEs is approximately double that of the three major JSEs, which is reasonable given the smaller size of their codebases. The `jitmut` engine attained the highest coverage for four of the seven JSEs, while `hybrjdon` and `mutation` outperformed the others by two JSEs, respectively. However, when considering the number of interesting samples that contributed to the increase in code coverage, `hybrjdon` used fewer samples for six out of the seven engines. For the three major engines — V8, JavaScriptCore, and SpiderMonkey — `hybrjdon` used approximately 83% of the interesting samples of the `mutation` engine, with numbers for V8

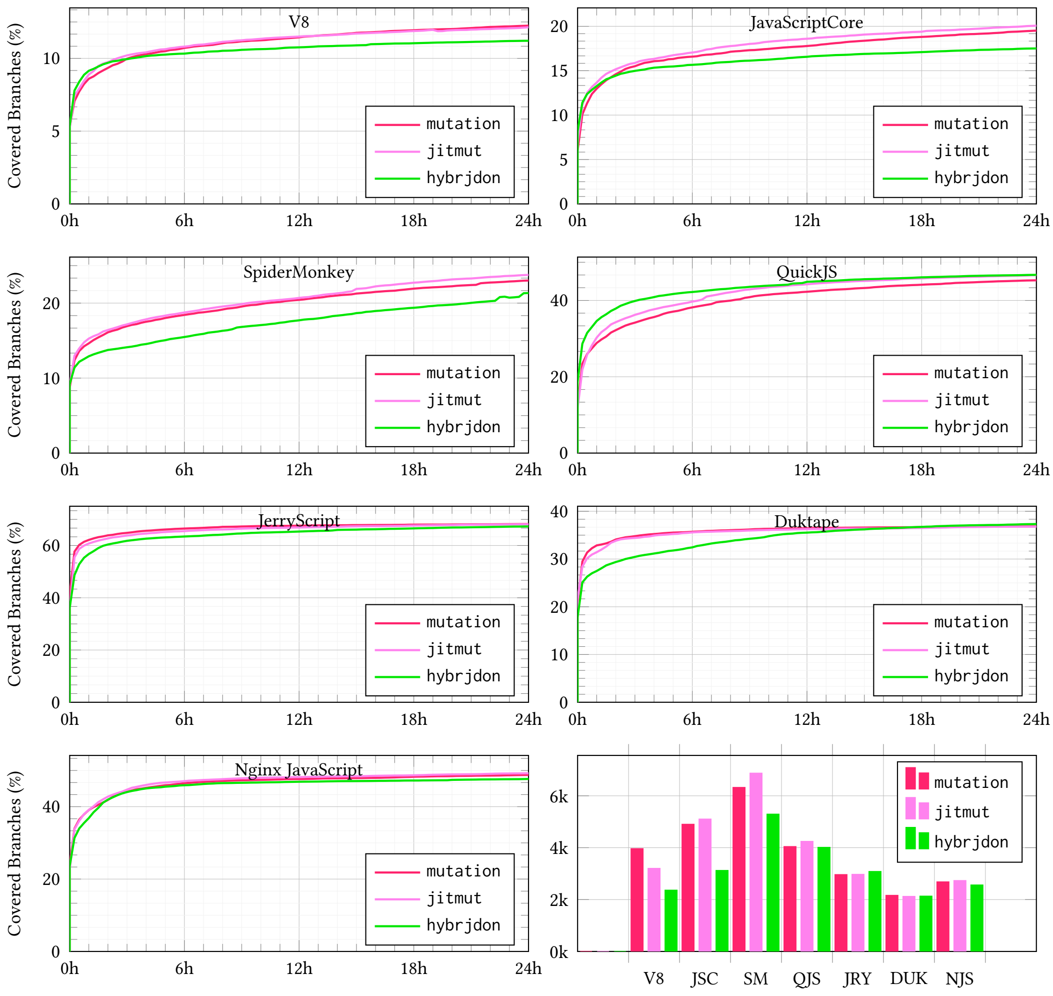

Fig. 8. The coverage trend observed within the 24-hour fuzzing. The last sub-figure highlights the number of *interesting* samples that contributed to the increase in code coverage shown in the previous sub-figures. All three engines achieved similar code coverage, with only negligible differences. However, the hybrjdon engine accomplished this with fewer samples for six out of the seven engines, especially on the three major engines.

and JavaScriptCore dropping to around ~59% and ~63%, clearly demonstrating the efficiency of hybrjdon-generated samples in increasing code coverage (see also the "CovEffi." row) even though we disabled many advanced features. We attribute this efficiency to the utilization of JIT-ops, which trigger diverse optimization paths and passes either in their JIT compilers (as seen in the three major JSEs) or within their interpreters (as in the other embeddable JSEs). It is also noteworthy that JoNM-empowered hybrjdon improves the valid ratio of the generated samples by over 7% due to its semantics-preserving feature.

We plot the total number of crashes found by each combination of fuzz engine and JSE in Figure 9. Interestingly, all three fuzz engines exhibited different performance across the two groups of JSEs. For the three major engines, none were able to identify any crashes within the 24-hour fuzzing budget. This is likely because of their frequent executions of Fuzzilli-based fuzzers; in fact, all

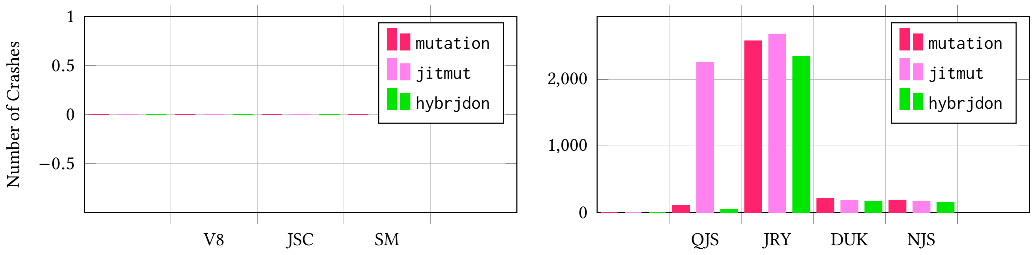

Fig. 9. The number of crashes triggered in each JSE during the 24-hour fuzzing. Instead of presenting the average numbers for the two instances, we provide the total counts in this figure. Notably, none of the fuzz engines could discover even one crash in the three major JSEs within a 24-hour fuzzing period while all of them triggered a similar number of crashes in three out of the four embeddable JSEs.

of them have integrated Fuzzilli into their build systems. However, all three engines caused the four embeddable JSEs to crash a significant number of times (over 100). Notably, they performed especially well with JerryScript, and the `jitmut` engine excelled with QuickJS. Upon manual inspection, we discovered that over 90% of the crashes reported in these two JSEs were flaky, as we could not reproduce them even after 100 additional attempts. In contrast, over 95% of the crashes reported in the other two JSEs — Duktape and Nginx JavaScript — were non-flaky. Despite performing comparably well, the `jitmut` engine demonstrated the best average performance, likely due to JIT-ops, particularly loops, which still triggered varied optimizations even within the interpreter. Regarding our mutators, `LI` was responsible for 58.8% of the crashes, with `SW` and `MJ` accounting for 23.8% and 17.2% of the crashes, respectively. `MD`, on the other hand, caused only 0.2% of the crashes, due to its requirement for a function call to be within a loop. We believe these findings suggest the significance and effectiveness of each mutator we have developed.

5.5 RQ2. Performance Overhead

The "Performance Statistics" section of Table 5 outlines the results. The performance overhead — whether in terms of throughput or execution cost — varies for each fuzz engine across different JSEs. Typically, the generated programs execute more slowly on the major JSEs compared to the embeddable ones, which is a reasonable result given their different application scenarios. Focusing on a single JSE, the `hybrjdon` engine consistently maintained a comparable or even lower performance overhead than the other engines. The only exception was observed in QuickJS, where the throughput of the `jitmut` engine was two/three times greater than that of the `mutation`/`hybrjdon` engines. This is possibly due to the significantly higher number of flaky crashes that it encountered, which caused their earlier returns. We were also surprised to find that both the `jitmut` and `hybrjdon` engines achieved higher throughput in some JSEs due to their tendency to insert loops. We attribute this performance boost to the disabling of advanced features that typically generate more complex code structures and the avoidance of inserting loops within loops.

5.6 RQ3. Mis-Compilations

During the seven-day fuzzing period, `Apollo` successfully discovered four mis-compilations, with three in JavaScriptCore and one in SpiderMonkey. Regarding the contributing mutators, two of the three mis-compilations in JavaScriptCore were identified by `SW`, while the remaining one and the one in SpiderMonkey were uncovered by `MJ`.

We have presented the SpiderMonkey mis-compilation in Figure 4. Below, we describe an intriguing mis-compilation in JavaScriptCore. This mis-compilation was triggered in revision b1d970cf based on a Fuzzilli-generated seed program. As with other figures, the highlighted code was inserted by `Apollo`'s `hybrjdon` engine. It is important to note that the loop inserted by `Apollo` is not neutral in this example, as the provided code is already an extensively reduced version.

```
const v0 = [0xAB0110, {}];
function f1(a2, a3) {
  if (a2 == "global") { v0[0] ^= a3;  return; }
  let v7 = v0[1][a2];
  if (v7 == undefined) { v7 = 0; }
  if (v7 < 50) { v0[0] ||= a3;  v0[1][a2] = v7 + 1; }
}
f1("global", 10569);
for (let i = 0; i < 921; i++) {
  try {
    f1("global", 14614);
    (1n)[1n];
    const o263 = { "maxByteLength": 536870912 };
    const v265 = new ArrayBuffer(10, o263);
    f1("global", 1047);
    const v273 = new DataView(v265);
    f1("global", 6363);
  } catch(e278) { print(e278); } finally {}
}
f1("global", 20614);
f1("global", 9736);
print(v0[0]);
```

The seed program contains a function `f1()` designed to update the value of `v0`. Indeed, this function is identical to `f3()` in Figure 4, as both are generated by `Apollo` to update the inserted checksum value (`v0`). In the seed example, `f1()` is called three times, and the checksum value is printed afterward. Additionally, `Apollo` inserts a loop that invokes `f1()` a total of 921×3 times, and also initializes a 512 MiB `ArrayBuffer` and a `DataView` instance. Since the program lacks any code related to randomness, it is expected to produce a consistent checksum value. Nevertheless, after the mutations applied by `Apollo`, JavaScriptCore's output alternates between two distinct checksums with an out-of-memory exception. Despite the developer's rejection to acknowledge it as a bug due to the exception, we insist that this indicates a *missed* JIT-optimization bug [61]. This is because `o263`, `v265`, and `v273` have never escaped to other scopes, which could be optimized out after each loop iteration. However, JavaScriptCore missed the optimization, leading to inefficient handling of the large memory allocation. To support this opinion, we ran the mutant on V8, SpiderMonkey, and QuickJS. All three engines produced the same checksum value without throwing any exceptions.

On the other hand, despite this promising result, we realized that `Apollo` did not find as many JIT-compiler bugs as `Artemis` (approximately six bugs per week). This is possibly because CSX and JoNM are generally designed, taking into account commonalities among LVMs but not considering their differences. For instance, LVMs that support dynamic types extensively utilize type-related speculations in JIT compilation to significantly reduce the cost of type guards, inline caching, and side-effect analysis among others before any access occurs and to enable more proper optimization paths and passes. To better support such LVMs, one may operate our approach on an extremely restricted subset of JavaScript, for example, asm.js to indicate high-performant optimizations by optimization tricks. In addition, type-related mutators need to be specially designed. We mark these improvements as one of our future work.

6 Discussions

We discuss the design choices in the implementation of JoNM, as well as the current capabilities and limitations of our implementations and the lessons we have learned from our experiments. Finally, we propose potential directions for interesting future work.

6.1 Design Choices

In this paper, we chose to realize CSX via a semantics-preserving, black-box strategy called JoNM.

Semantics-Preserving. While a non-semantics-preserving strategy may be effective in uncovering more crash bugs, it fails to detect mis-compilation bugs that are often considered more difficult, important, and harmful [3]. Additionally, it generates a vast mutation space that is challenging to sample systematically. In contrast, a semantics-preserving strategy like JoNM allows us to construct a tractable mutation space, capture mis-compilation bugs, and identify crash bugs.

Black-Boxing. Compared to white-box approaches, black-box strategies like ours are generally simpler and more portable, facilitating the rapid exposure of JIT-compiler bugs across LVM implementations. On the other hand, it would be beneficial to incorporate white-box techniques (e.g., designing directed-walk objectives using profiling data) for more effective realizations of CSX. This is an interesting and promising direction for future work.

6.2 Capabilities and Limitations

Capabilities. In theory, the ultimate goal of CSX is to exhaustively explore the compilation space of every real-world program and cross-validate the outputs. In practice, JoNM approximates CSX by taking into account engineering efforts, portability issues, and the trade-offs between the size of a program and its compilation space. Our tools have demonstrated their effectiveness, usefulness, and broad applicability by identifying numerous serious JIT-compiler bugs across all validated production JVMs while maintaining practical performance (Section 4), or by achieving comparable code coverage with a smaller number of samples while leading to a similar number of JSE crashes and capable of finding mis-compilations (Section 5).

Limitations. Currently, `Artemis` and `Apollo` only focus on mutating the exclusive methods of a program, which may miss some JIT-compiler bugs originating from built-in methods. Further, albeit acceptable, relying on loops — `LI`, `SW`, and `MJ` — can limit the performance of space exploration. A straightforward workaround is to set smaller compilation thresholds and larger/smaller `MIN_*/MAX_*` for validation. However, we decided to maintain the default thresholds as the discovered issues this way affect users more commonly; our one-week effort using this workaround did not yield anything interesting. One possible explanation is that this approach increases the number of methods subject to JIT compilation, which substantially reduces the overall compilation space. In contrast, our one-week effort using the default thresholds uncovered more than 154 discrepancies with `Artemis` (Section 4.3). Given our promising results, we anticipate finding many additional serious JIT-compiler bugs through a larger, more extensive testing campaign (e.g., using more time and additional processing cores). Lastly, neither of our tools currently supports concurrency and floating point numbers in JIT-compiler validation. These are generally considered difficult challenges in compiler testing and are expected to be addressed with more refined approaches [39].

6.3 Lessons Learned

Validating loop-related passes (e.g., Ideal Graph Building and Loop Vectorization) is crucial in optimizing compilers due to the complexity of loops that are typically the most time-consuming constructs. Despite extensive testing in the industry, we managed to discover over 15 bugs in loop-related passes of the validated JVMs, highlighting the importance of more attention to these passes. We also found that infinite loops are among the most effective triggers for JIT-compiler bugs. Even though recent advancements in validating loop optimizations in C compilers have been observed [39], limited research has been done in this area, particularly in optimizing JIT compilers. In addition, the management of LVM's memory, such as the allocation and cleanup of heap and stack memories, demands more attention, especially when associated with loops, as observed by our experiments on OpenJ9 and JavaScriptCore. Our experiment also shows the possibility of discovering missed optimization bugs by allocating large memories within loops. Finally, developers of LVMs should exercise caution when implementing analysis passes, such as Global Value Numbering, as their outcomes significantly impact subsequent transformation passes.

6.4 Future Work

Our approach enables several promising opportunities:

Effective and Efficient Mutators. It would be valuable to develop additional effective and efficient mutations to address the issues currently faced by our tools. Specifically, the focus should be on identifying: (1) Mutators that can enhance performance; (2) Uncommon traps that can trigger de-optimization across as many LVMs as possible. For the first case, one could create more de-optimization-oriented mutators, such as `MD`, to circumvent the side effects associated with loops. One could also integrate expressive loop idioms [39, 71] into construct synthesis. For the second, combining type information with speculative compilation could be effective in triggering de-optimizations, particularly for LVMs of dynamically typed languages like JavaScript and Python.

JoNM with Directed Walk. JoNM employs a random walk order, which is a stochastic sampling process over all possible program points. While this approach has already proven effective, future work could investigate alternative orders and mutation strategies more likely to identify interesting program points capable of triggering diverse optimization passes within JIT compilers. This might aid in exposing JIT-compiler bugs in earlier mutations, thereby accelerating the validation process.

Directed Walk with White-Box Information. Integrating white-box information into the objective of a directed walk also holds promise. For instance, one could record the JIT-trace coverage of the compilation space to guide the generation of uncovered JIT-traces. This could be achieved by leveraging or modifying the logging options of the JVM under test, such as HotSpot's `-XX:CompileCommand=log` or OpenJ9's `-Xjit:verbose`. Additionally, akin to coverage-guided fuzzing, one could track the code coverage of each JIT-trace and focus on those that cover new code or newly identified bugs.

Extending to Other LVMs. It would be promising to extend our work to validate other LVMs' JIT compilers, such as CPython. CSX and JoNM provide a general methodology, and `Artemis` and `Apollo` have demonstrated effectiveness in validating JIT compilers of JVMs and JSEs. In such LVMs, it is essential to specialize JoNM by taking into account types and type-related speculations.

7 Related Work

This work on JIT compiler validation lies at the intersection of LVM testing and compiler testing. Due to the importance of LVMs and compilers, the industry and the academia have invested substantial effort to improve their quality. This section surveys the most relevant work.

Testing JVMs. JVM testing is the most relevant thread of work; Table 6 shows a summary. Sirer et al. proposed a program generator for Java bytecode following a production grammar [57]; JavaFuzzer [21] and JFuzz [1] are two grammar-based random Java source generators; dexfuzz generates new bytecode tests by stochastically mutating existing seed programs in a domain-aware manner [29]; classfuzz leverages code coverage to guide bytecode mutation and generation [11]; classming focuses on smashing the control- and data-flow of the live bytecode area by inserting

Table 6. The most closely related work to ours on JVM testing. "# Reported Bugs": the number of bugs (if any) that the corresponding work listed in their paper; "Syntactical-Valid": whether the generated tests are syntactically valid; for mutation-based work (of which "Test Generation" is marked "M"), "Semantic-Preserving": whether their mutations preserve the seed's semantics; "JIT-Specific Testing": whether the work specifically aims at JIT compilers; and "Space Exploration": whether the work can thoroughly explore the compilation space modulo the LVM under testing.

		Venue	# Reported Bugs	Test Generation	Test Input Format	Test Method	Syntactical-Valid	Semantic-Preserving	JIT-Specific Testing	Space Exploration	To what extend can generated tests finally reach the JIT compiler?
Related JVM Testing Tools	Sirer et al. [57]	DSL '99	–	G	B	D	✓	–	×	×	Occasionally reach
	Yoshikawa et al. [77]	QSIC '03	–	G	B	D	✓	–	✓	×	Relies on AOT compilation
	JavaFuzzer [21]	–	–	G	S	D	✓	–	×	×	Occasionally reach
	JFuzz [1]	–	–	G	S	D	✓	–	×	×	Occasionally reach
	dexfuzz [29]	VEE '15	–	G	B	D	✓	–	✓	×	Relies on AOT compilation
	classfuzz [11]	PLDI '16	62	M	B	D	×	×	×	×	Occasionally reach
	classming [10]	ICSE '19	14	M	B	D	×	×	×	×	Occasionally reach
	JavaTailor [83]	ICSE '22	10	M	B	D	✓	×	×	×	Depends on ingredients
	JAttack [81]	ASE '22	6	G	S	D	✓	–	×	×	Depends on templates
	VECT [19]	ISSTA '23	26	M	S	D	✓	×	✓	×	Depends on ingredients
	JITfuzz [71]	ICSE '23	36	M	S	D	✓	×	✓	×	Depends on seeds and mutators
	SJFuzz [72]	ESEC/FSE '23	46	M	B	D	✓	×	×	×	Depends on seeds and mutators
	JOpFuzzer [27]	ICSE '23	41	M	S	P	✓	✓	✓	×	Depends on JVM options
	Jetris [82]	CCS '24	21	M	B	D	✓	×	×	×	Depends on ingredients
	LeJit [79]	FSE '24	15	G	S	D	✓	–	✓	×	Depends on extracted templates
	MopFuzzer [73]	ASPLOS '24	59	M	S	D	✓	×	✓	×	Depends on evoking mutators
	🦌 Artemis	SOSP '23	85	M	S	P	✓	✓	✓	✓	Reach by design

G: generation-based; **M**: mutation-based; **B**: `.class` bytecode; **S**: `.java` source-code;
D: differential testing: over multiple LVMs (or compilers), i.e., requiring other LVMs as references;
P: metamorphic testing: on the single LVM under testing, not requiring any other LVM.

control-flow altering bytecode sequences (e.g., goto, throw) into seed programs [10]; JavaTailor extracts five types of code ingredients from historical bug-revealing programs and synthesizes mutants by inserting them to seed programs [83]; VECT further clusters ingredients with similar semantics into the same group via code representations to improve testing performance [19]; Jetris generalizes bug-revealing patterns from historical program ingredients and instantiates them for testing [82]. JAttack and LeJit derive new tests by executing human-written skeletons and dynamically filling skeleton holes [79, 81]; SJFuzz devises new scheduling mechanisms for seeds and mutators in order to increase the possibility of finding inter-JVM discrepancies via differential testing [72]. These techniques, working at either the bytecode or source level, rely on differential testing over different JVMs to detect JVM bugs. In contrast, our approach differs in several aspects. First, our work introduces a novel metamorphic testing [9] approach: CSX explores the whole compilation space and differentially tests any two JIT-traces of a single program and a single LVM. Second, JoNM simulates this by differentially testing a seed program and its mutant inside a single LVM. Third, our approach specifically targets JIT compiler(s) in JVM, and JoNM is specially designed around JIT-relevant operations, i.e., loops, method calls, and uncommon traps.

There has been work on specifically testing JVM's JIT compilers. Yoshikawa et al. designed a random program generator [77]. They test the JIT compiler by directly AOT-compiling (ahead-of-time) the generated program using the JIT compiler under test, running the compiled machine code natively, and comparing the program outputs with several Java runtimes running bytecode. The tool dexfuzz applies the same comparison in their evaluation using different JVM backends [29]. These efforts belong to the traditional approach which compares the results of only a constant number of classical JIT-traces. However, CSX aims to explore the whole compilation space thoroughly. JITfuzz fuzzes JIT compilers guided by coverage and optimization-activating mutators [71]. JOpFuzzer explores and tests JIT compiler-related options [27]. MopFuzzer defines several mutators to trigger different JIT optimizations [73]. Versus `Artemis`, these tools require substantial expertise and human effort to understand different JVMs. Furthermore, JITfuzz and MopFuzzer are incapable of uncovering mis-compilations without differential testing and JOpFuzzer is limited to the number and functionality of exposed JIT compiler options. While some mutators from MopFuzzer resemble ours, they are not specifically designed for thorough JIT-compiler validation. However, we find their mutators intriguing and believe they can be well combined with ours.

Finally, work on other JVM aspects such as side channels of JIT compilation [4, 5], type systems [6, 7], garbage collections [47, 63], and JVM performance [36, 37] were also proposed recently. These have distinct scopes from our work.

Testing JSEs. JSEs are heavily tested for quality assurance and security aspects. LangFuzz can generate JavaScript programs by borrowing problematic code fragments from provided test suites [24]; Skyfire leverages a PCSG (probabilistic context-sensitive grammar) model for program generation [64]; Superion [65] and NAUTILUS [2] enhance AFL [78] by grammar-aware mutations; Fuzzilli fuzzes JSEs guided by coverage, based on a custom intermediate language [20]; CodeAlchemist can produce random yet semantics-preserving JavaScript programs by merging code bricks in a semantic-aware context [22]; DIE stochastically preserves the aspect (e.g., types and structures) during program mutation [53]; SoFi introduces an automatic repair strategy when a syntax- or semantic-invalid test is generated [23]; JEST checks the conformance between JavaScript specifications and engines [52]; FuzzFlow transforms a seed program into a graph-based intermediate representation and performs mutations on the graph [74]. There has been work leveraging deep learning techniques [32, 76] and large language models [16]. All the above work focuses on JSE interpreters or lack a specific focus on JSEs. For JIT compilers, JIT-Picker uncovers JIT-compiler bugs by differentially testing their interpreter and JIT compilers as the traditional approach [3]. FuzzJIT wraps existing code with a loop template to trigger JIT compilations [67]. Albeit similar to our `LI` mutator, FuzzJIT is specific to the loop template and unaware of the existence of the large compilation space. OptFuzz guides JIT-compiler fuzzing via optimization paths — the code path that starts from the entry statement of an optimization and ends at the statement performing the specific code transformation — based on the observation "entering an optimization ≠ triggering the optimization" [66]. This is a new perspective for thoroughly fuzzing internal optimizations inside JIT compilers and can be well integrated with `Apollo`, for example, by augmenting it with the coverage of optimization paths.

Testing Other LVMs. There has been work on testing other LVMs such as BPF [45, 46, 68], Ethereum [18], and Pharo LVM [55]. All these efforts in LVM testing have found many bugs. It would be promising to extend our approach to other LVMs like CPython and eBPF.

Testing Compilers. More research has concentrated on compilers. Program generators like Csmith [75] and YARPGen [38, 39] can produce random C programs. Creal generates new C programs by connecting code snippets from real-world programs [34]. MetaMut integrates large language models to automatically synthesize program mutators to facilitate compiler testing [51].

SPE applies skeletal program enumeration to generate C programs [80]. Alive [41] and Alive2 [40] attempt to validate optimizations. Another popular technique is EMI (Equivalence Modulo Input), a practical and effective idea that tests compilers by differential testing between a seed program and its EMI variants, finding thousands of bugs in GCC and LLVM [30]. Practical testing tools based on EMI exploit either the dead or live code region to derive EMI variants. Specifically, Orion randomly prunes the dead code from a seed program [30]; Athena enforces Markov Chain Monte Carlo (MCMC) to guide dead code deletion [31]; and Hermes inserts semantic-preserving code into the live area [59]. CLSmith adapts EMI to OpenCL [35].

Conceptually, JoNM belongs to the family of EMI techniques, especially live-code mutation. However, our mutations are fundamentally different from all proposed EMI work. First, JoNM aims to simulate CSX whose ultimate goal is to exhaustively explore the compilation space. Second, our mutations are applied on LVM's (optimizing) dynamic JIT compilers instead of static compilers, where the former heavily interacts with the corresponding LVM at runtime. Finally, our mutations are specially designed around JIT-ops which can trigger JIT/OSR compilation or de-optimization at runtime; this cannot be achieved by any EMI mutations proposed by far.

8 Conclusion

We have presented the novel concept of compilation space modulo LVM and an effective approach CSX for JIT-compiler validation. We proposed JoNM to approximate CSX, a lightweight, LVM-agnostic, and practical strategy leveraging JIT-ops for semantics-preserving mutations. We implemented the strategy as 🦌`Artemis` specifically for JVM, and our evaluation has led to 85 JIT-compiler bugs on three widely-used production JVMs: HotSpot, OpenJ9, and ART. We extended our approach to JSE and found four mis-compilations within only seven days via 🦅`Apollo`. We also discuss the lessons that we learned from our fuzzing campaign and several future directions toward extending CSX. We believe that the generality of CSX and JoNM likely makes them applicable and effective in other LVMs such as validating the JIT compilers of Python and .NET runtimes. This work introduces and opens this promising line of exploration.

Acknowledgments

We are grateful to the anonymous reviewers for their valuable feedback. This work was supported in part by the National Key R&D Program of China (Grant #2022YFB4501801), the Frontier Technologies R&D Program of Jiangsu (Grant #BF2024059), the Leading-edge Technology Program of Jiangsu Natural Science Foundation (Grant #BK20202001), and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

[1] ART. 2018. *JFuzz.* https://android.googlesource.com/platform/art/+/refs/heads/master/tools/jfuzz

[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Proceedings of the 2019 ISOC Network and Distributed System Security Symposium (NDSS '19).*

[3] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. Jit-Picking: Differential Fuzzing of JavaScript Engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22).*

[4] Tegan Brennan, Nicolás Rosner, and Tevfik Bultan. 2020. JIT Leaks: Inducing Timing Side Channels through Just-In-Time Compilation. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP '20).*

[5] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2020. JVM Fuzzing for JIT-Induced Side-Channel Detection. In *Proceedings of the 2020 ACM/IEEE International Conference on Software Engineering (ICSE '20).*

[6] Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Well-Typed Programs Can Go Wrong: A Study of Typing-Related Bugs in JVM Compilers. *Proc. ACM Program. Lang.* 5, OOPSLA (2021).

[7] Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. 2022. Finding Typing Compiler Bugs. In *Proceedings of the 2022 ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*.

[8] Craig David Chambers and David Michael Ungar. 1989. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the 1989 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '89)*.

[9] Tsong Yueh Chen, Shing Chi Cheung, and Shiu Ming Yiu. 1998. Metamorphic Testing: A New Approach for Generating Next Test Cases. *Department of Computer Science, The Hong Kong University of Science and Technology, Tech. Rep. HKUST-CS98-01* (1998).

[10] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep Differential Testing of JVM Implementations. In *Proceedings of the 2019 International Conference on Software Engineering (ICSE '19)*.

[11] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-Directed Differential Testing of JVM Implementations. In *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*.

[12] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. 1997. Compiling Java Just in Time. *IEEE Micro* 17, 3 (1997).

[13] David Curry. 2022. *Android Statistics (2022)*. https://www.businessofapps.com/data/android-statistics

[14] CVE. 2023. *Security Vulnerabilities (Memory Corruption)*. https://www.cvedetails.com/vulnerability-list/opmemc-1/memory-corruption.html

[15] Duktape. 2024. *Issue #273 - ES2015 Template Strings*. https://github.com/svaarala/duktape/issues/273

[16] Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. 2024. Fuzzing JavaScript Interpreters with Coverage-Guided Reinforcement Learning for LLM-Based Mutation. In *Proceedings of the 2024 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)*.

[17] Stephen J. Fink and Feng Qian. 2003. Design, Implementation and Evaluation of Adaptive Recompilation with on-Stack Replacement. In *Proceedings of the 2003 International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO '03)*.

[18] Ying Fu, Meng Ren, Fuchen Ma, Heyuan Shi, Xin Yang, Yu Jiang, Huizhong Li, and Xiang Shi. 2019. EVMFuzzer: Detect EVM Vulnerabilities via Fuzz Testing. In *Proceedings of the 2019 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*.

[19] Tianchang Gao, Junjie Chen, Yingquan Zhao, Yuqun Zhang, and Lingming Zhang. 2023. Vectorizing Program Ingredients for Better JVM Testing. In *Proceedings of the 2023 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*.

[20] Samuel Groß. 2018. *FuzzIL: Coverage Guided Fuzzing for JavaScript Engines*. Master's thesis. Karlsruhe Institute of Technology.

[21] Mohammad R. Haghighat, Dmitry Khukhro, Andrey Yakovlev, Nina Rinskaya, and Ivan Popov. 2018. *JavaFuzzer*. https://github.com/AzulSystems/JavaFuzzer

[22] HyungSeok Han, DongHyeon Oh, and Sang Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Proceedings of the 2019 ISOC Network and Distributed System Security Symposium (NDSS '19)*.

[23] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, and Wei Huo. 2021. SoFi: Reflection-Augmented Fuzzing for JavaScript Engines. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*.

[24] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 2012 USENIX Conference on Security Symposium (SEC '12)*.

[25] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the 1992 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '92)*.

[26] HotSpot. 2022. *Tiered Compilation*. https://github.com/openjdk/jdk11u-dev/blob/master/src/hotspot/share/runtime/tieredThresholdPolicy.hpp

[27] Haoxiang Jia, Ming Wen, Zifan Xie, Xiaochen Guo, Rongxin Wu, Maolin Sun, Kang Chen, and Hai Jin. 2023. Detecting JVM JIT Compiler Bugs via Exploring Two-Dimensional Input Spaces. In *Proceedings of the 2023 International Conference on Software Engineering (ICSE '23)*.

[28] Alexey Khrabrov, Marius Pirvu, Vijay Sundaresan, and Eyal de Lara. 2022. JITServer: Disaggregated Caching JIT Compiler for the JVM in the Cloud. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC '22)*.

[29] Stephen Kyle, Hugh Leather, Björn Franke, Dave Butcher, and Stuart Monteith. 2015. Application of Domain-Aware Binary Fuzzing to Aid Android Virtual Machine Testing. In *Proceedings of the 2015 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '15)*.

[30] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*.

[31] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '15)*.

[32] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. In *Proceedings of the 2020 USENIX Security Symposium (SEC '20)*.

[33] Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. 2023. Validating JIT Compilers via Compilation Space Exploration. In *Proceedings of the 2023 Symposium on Operating Systems Principles (SOSP '23)*.

[34] Shaohua Li, Theodoros Theodoridis, and Zhendong Su. 2024. Boosting Compiler Testing by Injecting Real-World Code. *Proc. ACM Program. Lang.* 8, PLDI (2024).

[35] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-Core Compiler Fuzzing. In *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*.

[36] David Lion, Adrian Chiu, Michael Stumm, and Ding Yuan. 2022. Investigating Managed Language Runtime Performance: Why JavaScript and Python are 8x and 29x Slower than C++, yet Java and Go can be Faster?. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC '22)*.

[37] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. 2016. Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems. In *Proceedings of the 2016 USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*.

[38] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA (2020).

[39] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. *Proc. ACM Program. Lang.* PLDI (2023).

[40] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of the 2021 ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*.

[41] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*.

[42] John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (1960).

[43] Microsoft. 2021. *Super Duper Secure Mode.* https://microsoftedge.github.io/edgevr/posts/Super-Duper-Secure-Mode

[44] MozillaSecurity. 2016. *funfuzz.* https://github.com/MozillaSecurity/funfuzz

[45] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Proceedings of the 2019 ACM Symposium on Operating Systems Principles (SOSP '19)*.

[46] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and Verification in the Field: Applying Formal Methods to BPF Just-In-Time Compilers in The Linux Kernel. In *Proceedings of the 2020 USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*.

[47] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-Performance Big-Data-Friendly Garbage Collector. In *Proceedings of the 2016 USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*.

[48] OpenJ9. 2020. *Recompilation.* https://github.com/eclipse-openj9/openj9/blob/master/doc/compiler/runtime/Recompilation.md

[49] OpenJ9. 2022. *Optimization Levels.* https://www.eclipse.org/openj9/docs/jit

[50] Oracle. 2023. *Autoboxing.* https://docs.oracle.com/javase/8/docs/technotes/guides/language/autoboxing.html

[51] Xianfei Ou, Cong Li, Yanyan Jiang, and Chang Xu. 2024. The Mutators Reloaded: Fuzzing Compilers with Large Language Model Generated Mutation Operators. In *Proceedings of the 2024 ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*.

[52] Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu. 2021. JEST: N+1-Version Differential Testing of Both JavaScript Engines and Specification. In *Proceedings of the 2021 IEEE/ACM International Conference on Software Engineering (ICSE '21)*.

[53] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP '20)*.

[54] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015).

[55] Guillermo Polito, Stéphane Ducasse, and Pablo Tesone. 2022. Interpreter-Guided Differential JIT Compiler Unit Testing. In *Proceedings of the 2022 ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*.

[56] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*.

[57] Emin Gün Sirer and Brian N. Bershad. 2000. Using Production Grammars in Software Testing. In *Proceedings of the 1999 Conference on Domain-Specific Languages (DSL '99)*.

[58] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph. D. Dissertation. Advisor(s) Bodik, Rastislav.

[59] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '16)*.

[60] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-Guided Program Reduction. In *Proceedings of the 2018 International Conference on Software Engineering (ICSE '18)*.

[61] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding Missed Optimizations through the Lens of Dead Code Elimination. In *Proceedings of the 2022 ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*.

[62] V8. 2023. *Maglev - V8's Fastest Optimizing JIT*. https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit

[63] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. 2022. MemLiner: Lining up Tracing and Application for a Far-Memory-Friendly Runtime. In *Proceedings of the 2022 USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*.

[64] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP '17)*.

[65] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *Proceedings of the 2019 International Conference on Software Engineering (ICSE '19)*.

[66] Jiming Wang, Yan Kang, Chenggang Wu, Yuhao Hu, Yue Sun, Jikai Ren, Yuanming Lai, Mengyao Xie, Charles Zhang, Tao Li, and Zhe Wang. 2024. OptFuzz: Optimization Path Guided Fuzzing for JavaScript JIT Compilers. In *Proceedings of the 2024 USENIX Security Symposium (SEC '24)*.

[67] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. 2023. FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler. In *Proceedings of the 2023 USENIX Security Symposium (SEC '23)*.

[68] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. 2014. Jitk: A Trustworthy in-Kernel Interpreter Infrastructure. In *Proceedings of the 2014 USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*.

[69] Webkit. 2014. *Introducing the WebKit FTL JIT*. https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit

[70] Webkit. 2016. *Introducing the B3 JIT Compiler*. https://webkit.org/blog/5852/introducing-the-b3-jit-compiler

[71] Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. 2023. JITfuzz: Coverage-guided Fuzzing for JVM Just-in-Time Compilers. In *Proceedings of the 2023 International Conference on Software Engineering (ICSE '23)*.

[72] Mingyuan Wu, Yicheng Ouyang, Minghai Lu, Junjie Chen, Yingquan Zhao, Heming Cui, Guowei Yang, and Yuqun Zhang. 2023. SJFuzz: Seed and Mutator Scheduling for JVM Fuzzing. In *Proceedings of the 2023 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23)*.

[73] Zifan Xie, Ming Wen, Shiyu Qiu, and Hai Jin. 2024. Validating JVM Compilers via Maximizing Optimization Interactions. In *Proceedings of the 2024 ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*.

[74] Haoran Xu, Zhiyuan Jiang, Yongjun Wang, Shuhui Fan, Shenglin Xu, Peidai Xie, Shaojing Fu, and Mathias Payer. 2024. Fuzzing JavaScript Engines with a Graph-based IR. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*.

[75] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*.

[76] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. 2021. Automated Conformance Testing for JavaScript Engines via Deep Compiler Fuzzing. In *Proceedings of the 2021 ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*.

[77] Takahide Yoshikawa, Kouya Shimura, and Toshihiro Ozawa. 2003. Random Program Generator for Java JIT Compiler Test System. In *Proceedings of the 2003 International Conference on Quality Software (QSIC '03)*.

[78] Michał Zalewski. 2016. *American Fuzzy Lop - Whitepaper*. https://lcamtuf.coredump.cx/afl/technical_details.txt

[79] Zhiqiang Zang, Fu-Yao Yu, Aditya Thimmaiah, August Shi, and Milos Gligoric. 2024. Java JIT Testing with Template Extraction. *Proc. ACM Softw. Eng.* 1, FSE (2024).

[80] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *Proceedings of the 2017 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17).*

[81] Zhiqiang Zhang, Nathan Wiatrek, Milos Gligoric, and August Shi. 2022. Compiler Testing via Template Java Programs. In *Proceedings of the 2022 International Conference on Automated Software Engineering (ASE '22).*

[82] Yingquan Zhao, Zan Wang, Junjie Chen, Ruifeng Fu, Yanzhou Lu, Tianchang Gao, and Haojie Ye. 2024. Program Ingredients Abstraction and Instantiation for Synthesis-based JVM Testing. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS '24).*

[83] Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-Driven Test Program Synthesis for JVM Testing. In *Proceedings of the 2022 International Conference on Software Engineering (ICSE '22).*