Validating JIT Compilers via Compilation Space Exploration

Cong Li
SKL for Novel Soft. Tech., Nanjing University
Nanjing, China
congli@smail.nju.edu.cn

Yanyan Jiang
SKL for Novel Soft. Tech., Nanjing University
Nanjing, China
jyy@nju.edu.cn

Chang Xu
SKL for Novel Soft. Tech., Nanjing University
Nanjing, China
changxu@nju.edu.cn

Zhendong Su
Department of Comp. Sci., ETH Zurich
Zurich, Switzerland
zhendong.su@inf.ethz.ch

Abstract

This paper introduces the novel concept of *compilation space*, which facilitates the thorough validation of just-in-time (JIT) compilers in modern language virtual machines (LVMs). The compilation space, even for a single program, consists of an extensive array of JIT compilation choices, which can be cross-validated for the correctness of JIT compilation. To thoroughly explore the compilation space in a lightweight and LVM-agnostic manner, we strategically mutate test programs with JIT-relevant, yet semantics-preserving code structures to trigger diverse JIT compilation choices. We realize our technique in `Artemis`, a tool for the Java virtual machine (JVM). Our evaluation has led to 85 bug reports for three widely used production JVMs, namely HotSpot, OpenJ9, and the Android Runtime. Among them, 53 have already been confirmed or fixed with many being critical. It is also worth mentioning that all the reported bugs concern JIT compilers, demonstrating the clear effectiveness and strong practicability of our technique. We expect that the generality and practicability of our approach will make it broadly applicable for understanding and validating JIT compilers.

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

Keywords: JIT compilers, JVMs, compilers, testing

ACM Reference Format:
Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. 2023. Validating JIT Compilers via Compilation Space Exploration. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3600006.3613140

1 Introduction

Modern programming language virtual machines (LVMs) are among the most critical and widely used system software ever developed. These LVMs typically *interleave* simple bytecode interpretation with dynamic, just-in-time (JIT) compiled code for improved performance. Well-known JIT compilers include HotSpot C1/C2, JavaScript V8 Turbofan, and eBPF JIT in the Linux kernel.

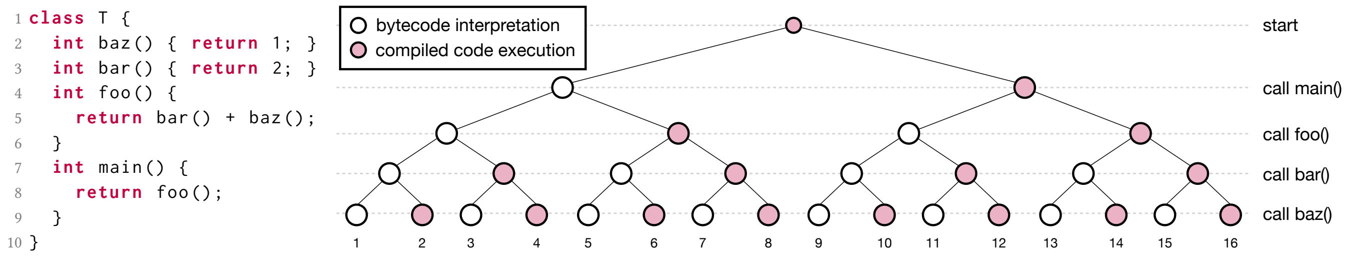
JIT compilers implement a broad range of nontrivial optimizations, making them one of the most complex components in modern LVMs. Like modern compilers [59], such complexity makes the JIT compiler a primary source of bugs in LVMs, leading to significant effort in LVM testing and validation by both academia and industry [11, 37, 39, 56, 58].

The challenge of validating optimizing JIT compilers lies in the tiered nature of JIT compilation, which has a relatively deep profiling-based "warm-up" process. First, generating syntactically and semantically valid test cases is challenging [10, 11, 21, 37, 64], resulting in few tests that can effectively exercise a JIT compiler deeply. Even for tests that can exercise the JIT compiler, they are insufficient to explore the vast compilation space thoroughly [39, 44, 45, 63]. As a result, most disclosed bugs are shallow and/or irrelevant to JIT compilers, such as early-stage parser or verifier bugs.

This paper introduces an approach to finding deep *JIT-compiler bugs* (crashes or mis-compilations) that do *not* manifest in the interpretation mode. Unlike existing techniques that simply treat JIT compilers as static compilers [1, 3, 26, 37, 56], to our knowledge, we are the first to exploit the *dynamic* nature of JIT compilers (interleaving between interpretation and compiled code execution) for thorough validation.

Compilation space modulo LVM. The key idea of this paper is to model the interleaving between the interpreter and JIT compiler by *compilation space modulo LVM* (compilation space for short). Assuming that a program makes n method calls where each method call can be independently *compiled* or *interpreted*, we get a compilation space consisting of 2^n possible JIT compilation choices. This naturally offers 2^n program versions plus a considerably strong test oracle asserting that all program outputs from the same compilation space should consistently remain the same.

The concept of compilation space significantly extends the testing space (implicitly) used by traditional approaches



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

Figure 1. The compilation space of a simple program, assuming each method call can be independently compiled or interpreted.

which consider only a few JIT compilation choices. Figure 1 depicts the compilation space of a simple program. The program has 4 method calls, and hence the compilation space consists of 16 possible JIT compilation choices. By contrast, the testing space of traditional approaches [26, 56] is composed of only a few JIT compilation choices like the fully-interpreted choice #1 and the fully-compiled choice #16. Additionally, it should be noted that running the program with any JIT compilation choice (#1–#16) should consistently return 3 for the `main` method. Otherwise, the LVM is considered to have a JIT-compiler bug.

Furthermore, the decision to switch from interpretation to JIT compilation (or vice versa) in practical LVMs like HotSpot could happen in the middle of a method, in addition to at method boundaries; this creates even larger compilation space and more validation opportunities. We provide further background on JIT compilers in Section 2 and formalize compilation space in Section 3.

Compilation space exploration. Different JIT compilation choices may lead to different optimization passes. Such diversity makes it beneficial to progressively explore every possible JIT compilation choice in the compilation space and cross-validate the equivalence of their outputs. We refer to this as *Compilation Space Exploration* (CSE).

A straightforward (and ideal) realization of CSE is modifying the LVM to expose an interface for complete control of JIT compilations. However, such modifications are likely incompatible with the internal assumptions of modern LVM implementations and evidently require substantial engineering effort for every single LVM to be validated.

To make CSE applicable to a wide range of JIT compilers with little effort, we further propose "JIT-Op Neutral Mutation" (JoNM), a novel, LVM-agnostic strategy for managing the LVM's decisions on when and how to JIT-compile specific code fragments. This is achieved through simple source-level mutations rather than complex LVM-level modifications.

The technical crux is how to use source-only modifications to control an LVM's compilation decisions. Our key observation is that CSE can be approximated by leveraging the mechanism for profile-guided compilation/optimization. Modern JIT compilers usually compile only a portion of a method and leave "uncommon traps" that fall back to the bytecode interpreter on cold paths. Hence, we can insert/remove loops or method calls (to enable/disable JIT compilation) and uncommon cold paths (to fall back or prevent from falling back to interpretation) to gain control over the interleaving between interpretation and compilation, e.g., jumping from the JIT compilation choice #1 to #6 in Figure 1.

Implementation. We developed `Artemis`, a simple implementation of JoNM that only manipulates method calls and loops, for validating three widely used production JVMs, namely HotSpot, OpenJ9, and the Android Runtime (ART). We reported 85 JIT-compiler bugs, of which 53 have been confirmed or fixed as of 10 April 2023. Among these, many are critical: 12 OpenJ9 bugs are classified as `blocker`, the most severe, release-blocking type of bug; 10 HotSpot bugs are marked as at least `P3`, major loss of function; and 13 OpenJ9 bugs are long latent across ≥4 major and many minor releases. Furthermore, our reported bugs stem from diverse errors in various JIT-compiler components such as 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"*. Section 4 presents our evaluation.

Contributions. Our main contributions are:

- We present the concept of compilation space and the CSE approach for thorough validation of JIT compilers.
- We propose JoNM, a novel, LVM-agnostic strategy to effectively approximate CSE from the source-code level.
- We show that even a basic implementation of JoNM can harness the power of CSE—85 JIT-compiler bugs are discovered in three widely used production JVMs: HotSpot, OpenJ9, and ART.

We make `Artemis`, along with all the reported bugs, publicly available via the following link to benefit the community and facilitate future research:

https://github.com/test-jitcomp/Artemis

2 Background and Illustrative Example

This section presents background on JIT compilers and a concrete HotSpot bug to motivate and illustrate our approach.

2.1 JIT Compilation

Compilation is costly. An LVM should carefully balance the cost of JIT compilation and its performance benefits. To make the start-up fast, a typical LVM boots in the interpretation mode and incrementally compiles the program [8, 12, 25, 36]. The LVM monitors a program's control flow (e.g., method calls and loop back-jumps) by profiling counters, and triggers a background JIT compilation when a counter reaches its threshold, a.k.a, being *hot*. The unit of JIT compilation can be either a method (method-JITs, e.g., HotSpot C1) or a code block like a loop (tracing-JITs, e.g., HotSpot C2).

JIT-compiled code may be active in the call stack for a long duration. Modern LVMs support *On-Stack Replacement* (OSR) to replace the interpreted bytecode stack frame with a native stack frame, which is also termed OSR compilation [15]. JIT optimizations may also make speculative assumptions on the program behavior to achieve good performance, e.g., assuming a code block is unreachable (and thus need not be compiled). Such assumptions are checked at runtime: Whenever any is violated, the program hits an *uncommon trap*, and the LVM falls back to the interpreter for a *de-optimization* [22]. Finally, JIT compilers may support leveled/tiered optimization, with more aggressive optimizations being carried out for already compiled code [23, 42]. All these mechanisms[1] highlight that a JIT compiler may switch back and forth between the interpreter and compiled code.

2.2 Illustrative Example

Given a seed program, `Artemis` validates JIT compilers by inserting neutral, semantics-preserving synthesized program constructs, yielding equivalent mutants with different JIT compilation choices. Figure 2 presents such a mutated test JDK-8288975 that causes HotSpot to mis-compile. It was detected at OpenJDK 11.0.15 (revision `f915a327`) but also affects JDK 17 and 20. The seed is generated by JavaFuzzer [18] and the (highlighted) mutant is derived by `Artemis`.

The seed itself calls `T.g()` only six times (by invoking `T.p()` twice at Line 29) and no JIT-compilation threshold is ever reached. Existing LVM testing techniques like JavaFuzzer intentionally try to avoid lengthy loops, otherwise most generated seeds would require excessive amounts of time to execute. To explore the compilation space, `Artemis` attempts to guide the JIT compiler to partially compile certain code segments by:

1. Pre-invoking `T.o()` for 9,676 times (Lines 21–22). To make this change semantics-preserving, `Artemis` also inserts a control flag `z` and a prologue to `T.o()` to return early on calls from Line 22.
2. Heating up `T.g()` by introducing a plain loop at Line 9 to try inducing higher-level JIT optimizations.

```
 1 class T {
 2   boolean z = false; byte l = 0;
 3   void g() {
 4     // (some omitted lines...)
 5     for (int m : k) {
 6       // (some omitted lines...)
 7       switch ((m >>> 1) % 10 + 36) {
 8       case 36:
 9         for (int w = -2967; w < 4342; w += 4);
10         // (some omitted lines...)
11         l += 2;
12       case 40: break;
13       case 41: k[1] = 9;
14       }
15     }
16   }
17   void o() { if (z) { return; } g(); }
18   void p() {
19     for (int q = 2; q < 5; ++q) {
20       z = true;
21       for (int u = 0; u < 9676; u++)
22         o();
23       z = false;
24       o();
25     }
26     System.out.println(l);
27   }
28   public static void main(String[] q) {
29     T t = new T(); t.p(); t.p();
30   }
31 }
```

Figure 2. JDK-8288975 triggers a mis-compilation in HotSpot. JavaFuzzer generates the seed, while `Artemis` inserts the highlighted code snippets. The code is reduced by Perses [52] and C-Reduce [48], and cleaned up for brevity.

Despite their simplicity, our mutations trigger nontrivial JIT compilations in HotSpot, leading to a drastically different JIT compilation choice: `T.o()` is first complied at the `L3` optimization level by HotSpot C1 and is further recompiled at the `L4` level by HotSpot C2. The call to `T.o()` also triggers a de-optimization when being called at Line 24 because HotSpot C2 speculatively assumes `z == true` at Line 17. The loop at Line 9 is OSR-compiled by HotSpot C2 at `L4` level speculating `w < 4342` and is de-optimized when the loop exits. `T.g()` is also JIT-compiled at the `L4` level.

Consequently, HotSpot mis-compiles the mutant and outputs a different `T.l` from the seed. The root cause is that the Global Code Movement (GCM) pass incorrectly moves a memory-writing instruction (`storeI`) from an outer loop to an inner loop because their estimated frequencies are the same. However, in fact, the inner loop executes three more iterations than the outer loop. To fix this, the developers prevented this pass from moving memory-writing instructions into loops deeper than their home loops.

[1] LVMs may have their own mechanisms and policies for JIT compilation, OSR, and de-optimization. However, these concepts remain valid throughout the discussions of the whole paper.

3 CSE and The `Artemis` Implementation

This section provides a rigorous description of compilation space (Section 3.1) and CSE (Section 3.2) and explains how JoNM (Section 3.3) and `Artemis` (Section 3.4) work.

3.1 Compilation Space modulo LVM

Profiling counters are the key to JIT-compilation control. This paper measures the hotness of such counters by their *temperatures*.

Definition 3.1 (Thresholds)**.** To facilitate multi-level compilation, an LVM has N compilation thresholds $0 \le Z_1 \le Z_2 \le \cdots \le Z_N \le +\infty$. In this paper, we define $Z_0 = 0$ and $Z_{N+1} = +\infty$ for simplifying the description. These compilation thresholds divide the counter values into $N+1$ ranges: $[Z_i, Z_{i+1})$ where $0 \le i \le N$.

Definition 3.2 (Temperature)**.** After bytecode parsing, an LVM for each method m maintains a set of profiling counters $C_m = \{c_0, c_1, \ldots, c_M\}$ which are updated at runtime. Specifically, c_0 is the method counter, and $c_1, c_2, \ldots, c_M$ are control-flow (e.g., loop back-edge) counters. A counter c is said to have *temperature* $\tau(c) = \mathtt{t}_i$ if and only if

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

where the temperature $\tau(c)$ satisfies a total order, i.e., $\mathtt{t}_i < \mathtt{t}_{i+1}$ always holds for $0 \le i \le N-1$.

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 $\mathtt{t}_0$ means that it is being interpreted and a method with temperature $\mathtt{t}_{i>0}$ tells that it is being executed with compiled code optimized at the i-th level. For the latter case, a method-JIT implies that the entire method m was already JIT/OSR-compiled, while a tracing-JIT only hints that the hot loop being executed within m has been OSR-compiled.[2]

Code can be heated up by executing method calls and loops and be cooled down by hitting uncommon traps; this paper names them *JIT-relevant operations* (JIT-ops). Executing JIT-ops in a method m causes $\tau(m)$ to change over time. This paper uses *temperature vector* u_m^i to represent such temperature change when m is called at the i-th time. Essentially, the temperature vector mirrors the interleaving between m's interpretation and compiled code execution, reflecting how an LVM compiles and de-optimizes m when it is called for the i-th time. For example, we can infer from $u_m^i = \langle \mathtt{t}_0, \mathtt{t}_1, \mathtt{t}_0 \rangle_m^i$ that: m is interpreted when it is immediately called for the i-th time, but it is then heated up and compiled at level 1 by JIT/OSR compilation; however, it is finally de-optimized and re-interpreted until this method call is completed.

Therefore, a JIT compilation choice, which is hereafter termed a *JIT compilation trace* (JIT-trace) in this paper, of a program can be represented by a sequence of temperature vectors. A JIT-trace, akin to an annotated method call trace, reflects how an LVM executes (interprets or compiles) a program *method call by method call*. Note that every program comes with a default JIT-trace for every LVM; this is the one generated when running the program with all default JIT-compiler options. The following showcases an example JIT-trace of a program:

$$\begin{aligned}\varphi = \langle \mathtt{t}_1 \rangle^1_{\mathtt{T.main}} \to \langle \mathtt{t}_1 \rangle^1_{\mathtt{T.T}} \to \langle \mathtt{t}_1 \rangle^1_{\mathtt{T.f}} \to \langle \mathtt{t}_1 \rangle^1_{\mathtt{T.b}} \to \\ \cdots \to \langle \mathtt{t}_1 \rangle^{10}_{\mathtt{T.f}} \to \langle \mathtt{t}_0, \mathtt{t}_1, \mathtt{t}_2 \rangle^{10}_{\mathtt{T.b}} \to \\ \cdots \to \langle \mathtt{t}_0 \rangle^{100}_{\mathtt{T.f}} \to \langle \mathtt{t}_2 \rangle^{100}_{\mathtt{T.b}} \to \langle \mathtt{t}_0 \rangle^1_{\mathtt{T.c}}.\end{aligned}$$

It tells that (1) the first 10 calls to `T.b()` enable itself to be JIT-compiled at temperature $\mathtt{t}_2$, (2) all subsequent calls to `T.b()` directly execute the compiled code, and (3) all other methods are continuously interpreted until `T.main()` exits.

Considering that a JIT compiler can compile, optimize, and de-optimize a method at many theoretically[3] valid program points beyond the method boundaries (especially for tracing-JITs), executing a program with n method calls can generate $\Omega(2^n)$ possible JIT-traces with respect to an LVM.

Definition 3.3 (Compilation Space modulo LVM)**.** Given a program P and a language virtual machine $\mathtt{LVM}$, all JIT-traces that can be generated by $\mathtt{LVM}$ with respect to P constructs the compilation space of P modulo $\mathtt{LVM}$:

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

where $\mathtt{LVM}(P, \varphi)$ requires $\mathtt{LVM}$ to generate the JIT-trace φ first and then returns the program output after running P along with φ, or $\bot$ if $\mathtt{LVM}$ cannot generate φ.

3.2 Compilation Space Exploration

An LVM should always yield *the same program output* no matter which JIT-trace $\varphi \in \mathbb{S}_{\mathtt{LVM}}(P)$ is generated when executing program P. Therefore, the JIT compiler in $\mathtt{LVM}$ is buggy if we can find $\varphi_1 \ne \varphi_2 \in \mathbb{S}_{\mathtt{LVM}}(P)$ where

$$\mathtt{LVM}(P, \varphi_1) \ne \mathtt{LVM}(P, \varphi_2).$$

Ideally, we should exhaustively check every legitimate $\varphi \in \mathbb{S}_{\mathtt{LVM}}(P)$ to cross-validate the equivalence of their program outputs. We call this *Compilation Space Exploration* (CSE).

Possible realizations. A straightforward and ideal realization of CSE is to customize an LVM with a complete control of the interleaving between interpretation and JIT compilation. However, this requires considerable engineering effort and is likely to be incompatible with some LVM's internal assumptions because, (1) practical LVMs are specially designed to allow JIT/OSR compilation and de-optimization only at specific program points, which makes some theoretically valid JIT-traces invalid, and (2) the space for even a small program is vast, often difficult or even impossible to compute due to implicit built-in library method calls (e.g.,

[2]Suppose background compilations are disabled or not supported.

[3]Some program points are considered valid for JIT/OSR compilation and de-optimization, while are likely to be disallowed by a practical LVM for some specific reasons like performance considerations.

one `println()` call involves dozens of built-in method calls). Another issue with this realization is that it is LVM-specific, thus not portable.

A practical realization is to fuzz the JIT compiler-related options of an LVM like JOpFuzzer [24], but (1) this needs substantial expertise and manual work to understand every JIT-compiler option in order to generate valid JIT-traces, and (2) the space exploration capability is largely constrained by the number and effects of LVM options. In addition, the understanding of one LVM's options cannot be used by other LVMs, which renders this realization not portable. We experimented with this realization by randomly choosing compilation thresholds for every test program, but our one-week effort did not lead to any interesting findings. Our experiences also tell us that compiler developers are not willing to fix bugs resulting from rarely used LVM options. These motivated us to look for a new CSE realization.

3.3 JIT-Op Neutral Mutation

JIT-Op Neutral Mutation (JoNM) approximates CSE from the source-code level with the help of JIT-ops, while being lightweight, LVM-agnostic, and practical for any LVM (such as JVM, JavaScript engines, etc.).

We leverage the mechanism for profile-guided compilation/optimization that LVMs execute method calls and loops to enable JIT compilation and hit uncommon traps to de-optimize. Thus, we rely on JIT-op (method calls, loops, and uncommon traps) mutations to gain control over the interleaving between interpretation and JIT compilation, and explore the whole compilation space progressively as more mutants are generated.

In particular, given a seed program P, JoNM stochastically samples a corpus of methods in P to insert, delete, or modify the JIT-ops (i.e., method calls, loops, and uncommon traps) within them to derive $\mathcal{P}$, a set of P's mutants. JoNM guarantees that the mutations are *neutral* to P's semantics. Specifically, every generated mutant $P' \in \mathcal{P}$ is ensured to (1) produce a different JIT-trace from P (by JIT/OSR-compiling a distinct code segment or de-optimizing at a distinct program point), but (2) preserve the same program output as P's. In this way, the thorough exploration of P's compilation space modulo LVM can be approximated by running a sufficient number of P's mutants. Hence, a JIT-compiler bug exists in LVM if we can find $P' \in \mathcal{P}$ where

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

Note that we intentionally omit the JIT-trace argument and directly use LVM(P) when executing P by requiring LVM to generate the default JIT-trace to simplify the description.

Versus other realizations, JoNM has several advantages:

- *Lightweight and simple*: JoNM approximates CSE at the source level, thus requiring negligible manual effort to understand LVMs and no modifications to the LVMs.

Algorithm 1: JIT-compiler validation by `Artemis`

```
procedure Validate(VirtualMachine LVM, Program P)
    $R \leftarrow$ LVM($P$)            // Run $P$ with $P$'s default JIT-trace
    for $i \leftarrow 1 \ldots$ MAX_ITER do
        $P' \leftarrow$ JoNM($P$)
        $R' \leftarrow$ LVM($P'$)  // Run $P'$ with $P'$'s default JIT-trace
        if $R' \neq R$ then          // Discrepancies imply bugs
            ReportJITCompilerBug($P'$)
function JoNM(Program P)
    $P' \leftarrow P$
    foreach Method $m \in P'$.Methods() do
        if FlipCoin() then
            $\phi \leftarrow$ Random mutator from LI, SW, and MI
            $\rho \leftarrow$ Random program point within method $m$
            $L \leftarrow$ SynLoop($\phi, \rho$)
            $P' \leftarrow \phi$.Mutate($P', m, \rho, L$)
    return $P'$
```

- *LVM-agnostic and wide-applicable*: Since JIT-ops are typically similar among all implementations of the same type of LVMs (e.g., HotSpot and OpenJ9 for JVM, V8 and SpiderMonkey for JavaScript engines), one JoNM implementation can be used to validate the same types of LVM implementations (e.g., all JVM implementations).
- *Practical*: JoNM can generate mutants based on real-world programs and programs generated by program generators. Therefore, (1) any found bug is likely to impact real-world users/vendors, and (2) it can empower any given program generator with the ability of practical JIT compiler validation.

3.4 The `Artemis` Implementation

We implemented JoNM for validating JVM's JIT compiler as `Artemis` which focuses mainly on synthesizing *neutral loops* using two kinds of JIT-ops: method calls and loops.

Algorithm 1 describes the main process. For each seed P, `Artemis` attempts to mutate it (Line 4) and run the mutant P' with its default JIT-trace (Line 5) for MAX_ITER times (Line 3). Since the mutations are neutral, it reports a bug once there is an output discrepancy between P and one of its mutants (Lines 6–7). JoNM works on P''s *exclusive methods* (methods defined and overridden in P'). In particular, it stochastically (Line 11) selects a corpus of P''s exclusive methods (Line 10) and mutates through three predefined mutators (Line 12), i.e., *Loop Inserter* (LI), *Statement Wrapper* (SW), and *Method Invocator* (MI), at an arbitrary program point ρ (Line 13). The mutation leverages a synthesized loop L which could heat up m to a higher temperature at program point ρ (Line 14). Finally, the synthesized loop L is inserted into the program point ρ by the selected mutator ϕ (Line 15).

Loop synthesis. SynLoop (Algorithm 2) follows the paradigm of programming-by-sketch to synthesize L, i.e., it synthesizes a program by filling holes left in a predefined

Algorithm 2: Loop synthesis in the context of JoNM

```
1  function SynLoop(Mutator $\phi$, ProgPoint $\rho$)
2      $L \leftarrow \phi$.loop_skeleton    // Initialized as the skeleton
3      $V \leftarrow \rho$.Variables()    // Variable set available at $\rho$
4      $V' \leftarrow \emptyset$    // Saving reused variables in synthesis
5      foreach ExprHole $\hbar \in L$.expr_holes do
6          $L \leftarrow$ Substitute($L$, $\hbar$, SynExpr($\hbar, V, V'$))
7      foreach StmtsHole $\hbar \in L$.stmts_holes do
8          $L \leftarrow$ Substitute($L$, $\hbar$, SynStmts($\hbar, V, V'$))
9      foreach Variable $v \in V'$ do
10         $L \leftarrow$ Backup $v$; $L$; Restore $v$;
11     return $L$
12 function SynExpr(ExprHole $\hbar$, VarSet $V$, VarSet $V'$)
13     $T$ = GetType($\hbar$)
14     if $T$ is a primitive-alike type then
           /* Rule 1: return a random value with the primitive
              alike type $T$ within the type $T$'s domain range. */
           /* Rule 2: return a random variable $v \in V$ with the
              type $T$; meanwhile expand $V'$ by $V' \leftarrow \{v\} \cup V'$. */
15     else if $T$ is an array type then
           /* Rule: create an array with dimension $T$.dimen and
              random size; let each array element as an
              expression hole typed $T$.comp_type and fill them by
              SynExpr; return the created array finally. */
16     else if $T$ has a non-parameter constructor then
17         return T()
18     else
19         return null
20 function SynStmts(StmtsHole $\hbar$, VarSet $V$, VarSet $V'$)
21     $S \leftarrow$ Random statement skeleton
22     foreach ExprHole $\hbar \in S$.expr_holes do
23         $S \leftarrow$ Substitute($S$, $\hbar$, SynExpr($\hbar, V, V'$))
24     return $S$
```

skeleton [50, 62]. In this paper, we design three types of holes for a loop skeleton: *expression holes* (`<expr>`), *statement holes* (`<stmts>`), and *placeholders* (`<placeholder:*>`), where the first would be filled by a Java expression and the others by Java statements. Figure 3 presents the loop skeleton of every predefined mutator. We equip each skeleton's loop header with customizable MIN, MAX, and STEP to ensure triggering different JIT/OSR compilation levels on different JVMs.

Given a mutator ϕ and a program point ρ, SynLoop synthesizes a loop L by filling ϕ's loop skeleton leveraging variables V that are available at ρ. In particular, it first synthesizes an expression for each `<expr>` and a statement list for each `<stmts>`, respectively. Then, it substitutes L's holes with the corresponding, synthesized code (Lines 5–8). Note, it does not fill `<placeholder:*>`s; they are left to the corresponding mutator (i.e., by ϕ.Mutate). It also backs up the value of every reused variable in both syntheses by a set V' (Line 4) and restores their values afterward (Lines 9–10) because the synthesized code may update reused variables in V'.

```
for (int i=min(MIN,<expr>); i<max(MAX,<expr>); i+=STEP) {
  <stmts>;
}                                                 // LI.loop_skeleton
-------------------------------------------------------------------
boolean exec = false;
for (int i=min(MIN,<expr>); i<max(MAX,<expr>); i+=STEP) {
  <stmts>;
  if (!exec) { <placeholder:stmt>; exec = true; }
  <stmts>;
}                                                 // SW.loop_skeleton
-------------------------------------------------------------------
for (int i=min(MIN,<expr>); i<max(MAX,<expr>); i+=STEP) {
  <stmts>;
  P.m_ctrl = true; <placeholder:method>; P.m_ctrl = false;
  <stmts>;
}                                                 // MI.loop_skeleton
```

Figure 3. Loop skeletons of LI, SW, and MI. Symbols `<expr>`s and `<stmts>`s are expression and statement holes that should be synthesized when synthesizing loops, respectively; yet `<placeholder:*>`s are placeholders that should be substituted when the corresponding mutator is making mutations. Hyper-parameters MIN, MAX, and STEP are customizable.

Expression synthesis. SynExpr synthesizes an expression concerning the hole $\hbar$'s type T (Line 13):

- For primitive-like types including boxed/unboxed [43] primitive types and String, SynExpr either (1) generates a random value with type T or (2) reuses an existing T-typed variable $v \in V$. In the latter case, it also saves the reused variable v to V'.
- For array types, SynExpr first creates an array instance with the component type T.comp_type, the dimension T.dimen, and a random size for each dimension. It then fills the array by regarding each array element as an expression hole with type T.comp_type and recursively invokes SynExpr to synthesize an expression for each element. Finally, it returns the array.
- For reference types, SynExpr always creates a new object if there is a non-parameter constructor. Otherwise, a null is returned. Artemis does not reuse reference variables since access to their fields or methods is likely to update their values implicitly.

Statement synthesis. Instead of generating statements from scratch, Artemis collects a corpus of statement skeletons from HotSpot, OpenJ9, and ART's test suites, by following existing practices in LVM testing [19]. Each statement skeleton is a sequence of consecutive Java statements with `<expr>` holes only. SynStmts then randomly picks a statement skeleton (Line 21) and fuses an expression for each expression hole inside it (Lines 22–23).

In JoNM, `<stmts>` and statement skeletons are not a must. However, the synthesized loop L becomes far more diverse in terms of the control- and data-flow because of them. This makes L capable of triggering varied optimization passes in

JIT compilers of the validated JVM. Together with V', this also prevents L from being optimized away by the compiler.

Mutator's mutation. The synthesized loop L is finalized and P' is mutated by three mutators: Loop Inserter (`LI`), Statement Wrapper (`SW`), and Method Invocator (`MI`).

Loop Inserter. `LI.loop_skeleton` does not contain any `<placeholder:*>`, so it directly inserts L into program point ρ. Consequently, the loop would heat up m to be OSR-compiled at some compilation levels. Depending on the JVM, this may also bring an extra de-optimization when the loop exits.

Statement Wrapper. `SW` firstly replaces `<placeholder:stmt>` with the statement s right after ρ, then removes s from P', and finally inserts L at ρ. As a result, the statement s is wrapped by the synthesized loop, and the control- and data-flow at ρ are greatly altered. To avoid changing the semantics, `SW` guarantees the wrapped statement s is executed only once by introducing a control flag `exec` (Figure 3, `SW`, Line 5). Like `LI`, `SW` can bring OSR compilations (and perhaps de-optimizations depending on the JVM).

Note that the essential difference between `LI` and `SW` shows when they are applied to tracing-JITs: `SW` drives the wrapped statement and the inserted loop to be compiled together whereas `LI` exclusively focuses on the JIT compilation of the inserted loop. Therefore, they induce different control- and data-flow optimizations within the JIT compiler.

Method Invocator. In addition to OSR compilation, `MI` is designed to trigger JIT compilation by method calls. Specifically, `MI` first replaces `<placeholder:method>` (Figure 3, `MI`, Line 14) by a synthesized method call to m using `SynExpr` and the following skeleton (`<expr>`s are m's arguments)

```
m(<expr>, <expr>, ...);
```

Next, from all method calls to m in P', `MI` selects a random one and inserts the finalized L right before it. Such insertion drives JVM to JIT-compile m before the selected call.

Yet, introducing additional method calls to m may change the semantics. To avoid this, `MI` synthesizes another piece of code using `SynStmts`, `SynExpr`, and the following skeleton

```
if (P.m_ctrl) { <stmts>;  return <expr>; }
```

and inserts the synthesized code as the very first statement of m. The above skeleton involves a control variable `m_ctrl` which is introduced as a new class field. In L, `P.m_ctrl` is set to true before invoking m and set back to false afterward (Figure 3, `MI`, Line 14). Thus, running L causes the synthesized code to be executed only once, and m always *early returns* without executing any other statements.

Figure 2 provides a concrete example for `MI`. In this example, the method m is `T.o()`; the highlighted code at Lines 20–23 is our synthesized loop L which pre-calls `T.o()` for 9,676 times; and the method call `o()` at Line 24 is our picked call. To preserve the semantics, a control variable `z` is introduced to class `T` at Line 2 and set to `true` before pre-invoking `T.o()`. During pre-calls, our synthesized code highlighted at Line 17 is executed and early returns, leaving other statements of `T.o()` unexecuted. Later, `T.z` is set to false such that our picked call can execute as normal.

Other considerations. The performed mutations so far are not completely neutral because the collected statement skeletons may have unexpected behaviors like throwing exceptions. Thus, all three mutators—after their aforementioned mutator-specific mutations—apply the following three mutations as their final step: (1) rename every variable in L with a new name to avoid name conflicts, (2) replace `System.out` and `System.err` by a `PrintStream` that prints nothing before executing L, and restore their values afterward to avoid unexpected output, and (3) catch and discard every exception likely to be thrown by L.

Implementation details. We have implemented `Artemis` in ~3,000 lines of Java and ~2,000 lines of Python. It relies on the Spoon framework [46] to parse the Java source code and enable skeleton definition and instantiation capabilities. We also extracted a total of 7,823 statement skeletons by parsing HotSpot, OpenJ9, and ART's existing test suites, following existing practices in LVM testing [19].

4 Evaluation

This section describes the evaluation of our approach by applying `Artemis` to validate three widely used production JVMs: HotSpot, OpenJ9, and ART. Highlights of our results as of 10 April 2023 are as follows:

- **Many detected bugs**: We have reported 85 bugs, of which 53 have been confirmed or fixed by the corresponding developers. The 85 bugs affect all the three JVMs with at least 16 bugs for each JVM.
- **All JIT-compiler bugs**: All our reported bugs manifest themselves only when JIT compilers are enabled; otherwise, these bugs disappear.
- **Many serious bugs**: Many of the reported bugs are critical, blocking the development of the next release or being long-latent across several major releases.

We believe that (1) the quantity and quality of our reported bugs have demonstrated the clear effectiveness of our approach in JIT-compiler validation, and (2) at least 16 bugs per JVM show the general applicability of our approach.

4.1 Evaluation Setup

JVMs and versions. Our evaluation focused on three widely used production JVMs: HotSpot, OpenJ9, and ART. We chose HotSpot and OpenJ9 based on their popularity by following existing work [10, 11, 64]. ART was selected as our subject because of its tremendous user base [13]. The open-source nature, openness, and activeness of their bug systems also help us track bugs, discussions, and fixes. For HotSpot and OpenJ9, we chose JDK 8, 11, and 17 because they are long-term supported (LTS). ART is excluded from choosing JDK

Table 1. Statistics of 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

versions because it does not support class bytecode directly.[4] For each selected JVM, we built its latest trunk and validated it with (1) background compilation (if supported) disabled and (2) 1 GiB Java heap memory. We did not choose the latest stable releases since their bug fixes are only available in subsequent stable releases. Such a long time gap (as well as the concurrency from background compilation) hinders us from distinguishing whether a newly detected bug duplicates an existing one. Finally, our evaluation mainly focused on the x86_64 Linux platform.

Seed programs. We used JavaFuzzer [18], a random Java program generator, to generate seed programs for `Artemis` because JavaFuzzer-generated programs are generally complex, providing rich opportunities for `Artemis` to mutate. Moreover, our experience tells us that JavaFuzzer-generated code can be effectively reduced by combining Perses [52] and C-Reduce [48]. However, it should be noted that `Artemis` is agnostic to seed programs, which means `Artemis` can be incorporated with other Java program generators or even real-world programs. We did not use them in our evaluation mainly because it typically takes a long time to reduce the tests generated by them.

Synthesis parameters. Our experience suggests that eight mutants appear to strike a good cost/effectiveness balance for exploring the compilation spaces of the seed programs generated by JavaFuzzer after hundreds of attempts; thereby in our evaluation, we set `MAX_ITER` to 8 to simulate exploring eight JIT-traces for each seed program. Since different JVMs define different default compilation thresholds, to ensure JIT and OSR compilations, `MIN` and `MAX` are set accordingly: 5,000 and 10,000 in HotSpot/OpenJ9 while 20,000 and 50,000 in ART. We let `Artemis` pick a random `STEP` ranging from 1 to 10 when synthesizing loops.

4.2 Quantitative Results

Numbers of bugs. We have filed in total 85 bugs for the three JVMs, including 32 in HotSpot, 37 in OpenJ9, and 16 in ART. Among them, 17 bugs were discovered in method-JITs (i.e., HotSpot C1 and ART OptimizingCompiler) and 68 bugs in tracing-JITs (i.e., HotSpot C2 and OpenJ9's JIT). The first half of Table 1 presents the status of them. As of 10 April 2023, 53 have already been confirmed and 26 have been fixed. We recognize a reported bug as "Confirmed" if the corresponding JVM developers can reproduce the bug in their settings. Otherwise, we leave them in the "Reported" category regardless of whether we have a complete crashing log for reproduction and diagnosis. Although we ensured that all reported bugs behave with different symptoms (e.g., stacktraces), two bugs for ART and five for OpenJ9 still stem from the same root causes as some bugs that we had reported previously; we also reported eight unique HotSpot bugs duplicating those reported by other developers or users, showing that `Artemis` can find bugs that common users actually encounter in development. We categorize all these as "Duplicate".

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

Mis-compilation. JIT compiler incorrectly compiles the program, which incurs a semantic disagreement between bytecode and compiled code, i.e., running them yields different program outputs. This is likely due to (1) bytecode compilation, (2) upper-level optimization, or (3) de-optimization.

Crash. JVM crashes either when compiling the code or when executing the compiled code. The symptoms are various, e.g., segmentation faults, and assertion failures.

Performance issue. Executing the compiled code causes JVM to be obviously slower than interpreting the bytecode. This is typically user-perceivable and there are chances that the JVM process is finally killed by the operating system.

The second half of Table 1 classifies our reported bugs into these three categories. More than 20% are mis-compilations, the most interesting and hard-to-detect bugs [3]. We found only one performance bug in which the HotSpot process running the test is killed on Ubuntu while it runs noticeably slow on Windows. Even though we have detected many mis-compilations on both OpenJ9 and ART, HotSpot is an exception possibly because HotSpot, as the most prevalent JVM, is much more mature than other JVMs.

Importance of bugs. It is worth mentioning that all the reported bugs are JIT-compiler bugs that are otherwise hidden by the bytecode interpreter if JIT compilers are disabled.

In addition, quite a few of the bugs are deemed serious. In particular, 12 out of the 37 OpenJ9 bugs were tagged as `blocker`, the most severe, release-blocking type of bug. We also detected 10 out of the 32 HotSpot bugs marked as ≥`P3` (major loss of function). There have been 13 long latent OpenJ9 bugs across ≥4 major and many minor releases, escaping the testing campaigns by earlier and contemporary tools. The developers were surprised by the effectiveness of

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

Table 2. Affected JIT compiler components by reported JIT compiler crashes in HotSpot and OpenJ9. Columns "#" are the number of JIT compiler crashes affecting the corresponding component. "Code Execution" represents that the crashes happen when JVMs are executing the compiled code. "Other JIT Components" includes JIT-INT interaction, synchronization, etc. "Garbage Collection" indicates that the JIT compiler triggers a crash in the garbage collector.

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 [41]	1
Code Generation, C2	3	Other JIT Compone.	6
Code Execution, C2	3	Garbage Collection	13

our effort and even asked "*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 JIT compiler components. Bugs we have reported are diverse, affecting various JIT compiler components as shown in Table 2. Because it is difficult to recognize which components are affected for mis-compilations and performance issues (if not yet fixed), we only consider crashes. We also exclude JVMs having fewer than 10 crashes because their results are not considered reliable.

HotSpot. The 30 crashes affect 8 C1/C2 components, where most are of C2. This is reasonable because C2 is considered far more complicated than C1 with more aggressive optimizations. 29 out of the 32 crashes happen when C1 or C2 is compiling and the other three (i.e., column "Code Execution") happen when executing the compiled code. Specifically, the most affected component is ideal loop optimization, followed by global value numbering and ideal graph building.

Table 3. Mutation cost of `Artemis` in seconds. Row "Single-run" refers to the cost of generating a single mutant via `Artemis`. Row "Large-scale" is the cost when `Artemis` is booted only once but generates a magnitude of mutants.

	Mean	Median	Min	Max
Single-run	1.65	1.68	0.76	2.01
Large-scale	0.16	0.16	0.06	2.19

OpenJ9. The affected components of OpenJ9 are different from HotSpot. Specifically, the 28 crashes affect ≥8 JIT compiler components, where 26 of them happen when OpenJ9's JIT compiler is compiling the code, and the other two happen when executing the compiled code. To our surprise, most crashes occur inside the garbage collector. We discussed these crashes with OpenJ9's developers and learned that these are indeed JIT-compiler bugs because it is the JIT compiler that corrupts the heap memory, causing the garbage collector to crash. Furthermore, if these heap memory corruptions are mishandled, they can result in serious exploitable security vulnerabilities [14], suggesting that JIT-compiler bugs pose a significant threat as they can impact various JVM components beyond the JIT compiler itself. For JIT compiler components, the most affected are global value propagation and code generation.

Mutation cost. Given a seed program, the cost of `Artemis` to generate one mutant is low. On average, it took ~1.65 seconds for `Artemis` to complete both (syntax and semantic) source parsing and loop synthesis, where the former cost ~0.67 seconds and the latter ~0.88 seconds. In a setting of large-scale JVM fuzzing, where `Artemis` and its dependent skeleton engine Spoon [46] are booted only once but driven to generate numerous mutants for quantities of seed programs, the mutation cost is negligible: it took only ~157 milliseconds to mutate a seed program on average. Table 3 shows more statistics, in which the relatively large cost "2.19" occurs only at the very first mutation (when being booted).

4.3 Comparative Study and Throughput

To further investigate the effectiveness of our approach, we conducted a comparative study with the traditional approach to show the power of exploring more JIT-traces. We also measured the throughput of `Artemis` at the same time.

In this study, we reused the synthesis parameters mentioned in Section 4.1 and chose OpenJ9 (JDK 11, revision `4ca209b5`) as the validation target. We used JavaFuzzer as the seed generator. For each seed, we first ran it once with its default JIT-trace in OpenJ9. Next, we ran it again by forcing every method to be JIT-compiled before their first calls by the `-Xjit:count=0` OpenJ9 option, regarding its JIT compiler as a static compiler like traditional approaches [26, 56]. Then, we mutated the seed 8 times using `Artemis` and ran each

Table 4. Comparative study between CSE and the traditional approach. The first two columns read the number of seeds and mutants generated. Columns "CSE" and "Tra." list the number of seeds for which the corresponding approach can spot output discrepancies. Column "Both" is the number of seeds that both approaches can find output discrepancies.

#Seeds	#Mutants	CSE	Tra.	Both
42,559	340,472	154	21	16

mutant with its default JIT-trace. Finally, we compared the program outputs and counted the number of seed programs leading to output discrepancies.

We conducted the study on an AMD server with a Ryzen Threadripper 3990X 64-core processor for 7 days. To demonstrate that `Artemis` works well even on commodity machines, we enabled 16 of the 64 cores. During this process, we discarded seed programs or mutants that could not finish within 2 minutes. Table 4 presents the results.

Results. During 7 days, `Artemis` drove JavaFuzzer to generate 42,559 seeds and mutated them 340,472 times. Among these seeds, `Artemis` successfully steered 154 to trigger discrepancies, where 89.6% (138) cannot be triggered simply by comparing the default and fully-compiled JIT-trace. There are 5 seeds for which `Artemis` was unable to trigger any difference within 8 mutants. We inspected them in detail and found that they involve JIT/OSR compilations of built-in method calls, which is beyond `Artemis`' capability. We will discuss this further in Section 4.5.

Throughput. In this process, `Artemis` invoked OpenJ9 ≥383,031 times, with a throughput of ≥0.63 OpenJ9 invocations per second. That being said, `Artemis` can test a program in ~15s (including 9 source-bytecode compilations and 10 OpenJ9 invocations). Most CPU time is spent on source-bytecode compilation and executing the synthesized loops. Considering that (1) `Artemis` relies mainly on loops due to which the mutant often takes long (typically dozens of seconds) to finish, and (2) we only enabled 16 cores during evaluation, we believe that this throughput is practical.

4.4 More Examples

`Artemis` is fruitful in finding diverse bugs such as segmentation faults (SIGSEGV), fatal arithmetic error (SIGFPE), emergency abort (SIGABRT), assertion failures, mis-compilations, and performance issues. More examples are available at `Artemis`' website which also contains complete references to all reported JIT-compiler bugs.

4.5 Discussions

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

Semantics-preserving. Although a non-semantics-preserving strategy may help reveal more crash bugs, it not only is incapable of detecting mis-compilation bugs—which are deemed more difficult, important, and harmful [3]—but also introduces a huge mutation space that is more difficult to systematically sample. In contrast, a semantics-preserving strategy like JoNM helps construct a tractable mutation space, capture mis-compilation bugs, and also find many crash bugs.

Black-box. Versus white-box realizations, black-box ones like ours are in general simpler and more portable, helping quickly expose JIT-compiler bugs in any LVM. On the other hand, it would be fruitful to integrate white-box techniques (e.g., guiding mutation by profiling data) for more effective realizations of CSE, which we consider interesting and promising future work.

Capabilities and limitations. In theory, the ultimate goal of CSE is to exhaustively explore the compilation space of every real-world program and cross-validate the equivalence of their outputs. In practice, JoNM approximates CSE by taking into consideration the engineering effort, portability issues, and trade-offs between the size of a program and its compilation space. `Artemis` has confirmed its effectiveness, usefulness, and broad applicability by finding many serious JIT-compiler bugs in every validated production JVM (Section 4.2) with practical throughput (Section 4.3).

Currently, `Artemis` only focuses on mutating the exclusive methods of a program; this may miss some JIT-compiler bugs caused by built-in methods. Further, albeit acceptable, relying on loops limits the throughput of space exploration. A simple workaround is to set smaller JIT compilation thresholds and smaller `MAX` in validation. However, we decided to adopt the default thresholds as the discovered issues this way affect users more commonly and our one-week effort using this workaround did not yield anything interesting. A possible reason could be that this workaround increases the number of methods to be JIT-compiled, which considerably reduces the compilation space. As a comparison, our one-week effort using the default thresholds led to more than 154 discrepancies (Section 4.3). Considering our very positive results, we expect to find many additional serious JIT-compiler bugs when running a larger, more extensive testing campaign (e.g., using more time and cores). Finally, `Artemis` does not currently support concurrency and floating point. These are generally deemed difficult challenges in compiler testing and are expected to be addressed with finer-grained approaches [33].

Future work. CSE enables several promising opportunities for future work. First, it would be interesting to devise additional effective and efficient mutations to mitigate the issues that `Artemis` currently faces. Specifically, the key is to find (1) mutations that can help improve throughput and (2) general uncommon traps that can take effect in as many LVMs as possible. Second, JoNM applies stochastic sampling over all possible program points. Future work could explore other mutation strategies capable of finding interesting program points that are more likely to trigger diverse optimizations. This may help expose JIT-compiler bugs in early mutations, accelerating the validation process. Third, integrating white-box techniques and expressive loop idioms [33, 58] into loop synthesis is also promising. For example, we can record the coverage of the compilation space and guide `Artemis` to generate uncovered JIT-compilations; this can be accomplished by leveraging/hacking the logging options of the JVM such as HotSpot's `-XX:CompileCommand=log`, OpenJ9's `-Xjit:verbose`. Finally, it would be interesting to extend our work to validate other LVMs such as JavaScript engines. This is promising because CSE and JoNM have offered a general, high-level methodology, and `Artemis` has been shown effective in finding many critical JIT-compiler bugs in three widely used production JVMs.

5 Related Work

This work on JIT compiler validation and testing lies at the intersection of LVM testing and compiler testing. This section surveys many closely related work.

Testing JVMs. JVM testing is the most relevant thread of work, especially to `Artemis`; Table 5 shows a summary.

Sirer et al. proposed a program generator for Java bytecode following a production grammar [49]; JavaFuzzer [18] 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 [26]; 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 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 [64]; JAttack derives new tests by executing human-written skeletons and dynamically filling skeleton holes [63]. These techniques, working at either the bytecode or source level, rely on differential testing over different JVMs to detect JVM bugs. By contrast, our approach differs in several aspects. First, our work introduces a novel metamorphic testing [9] approach: CSE explores the whole compilation space and cross-validates any two JIT-traces of a single program on a single LVM. We aim to control the interleaving of JIT compilation and interpretation. Second, JoNM approximates this by cross-validating 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 [61]. They test the JIT compiler by directly AOT-compiling (ahead-of-time) the generated program using the JIT compiler under test, running the compiled 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 [26]. These efforts belong to the traditional approach which compares the results of only a constant number of classical JIT-traces, simply treating JIT compilers as static compilers. However, CSE explores the dynamic nature, aiming to explore every interleaving (i.e., JIT-trace) in the compilation space and cross-validate the equivalence of their outputs. JITfuzz fuzzed JIT compiler guided by coverage and optimization-activating mutators [58]. JOpFuzzer explored and tested JIT compiler-related options [24]. Versus `Artemis`, both tools require substantial expertise and human effort to understand different JVMs. Furthermore, JITfuzz is incapable of uncovering mis-compilations without differential testing and JOpFuzzer is limited to the number and functionality of exposed JIT compiler options. Nevertheless, these efforts, orthogonal to ours, are promising to be integrated with CSE.

Finally, work on other JVM aspects such as side channels of JIT compilation [4, 5], type systems [6, 7], garbage collections [40, 53], and JVM performance [30, 31] were also proposed recently. These have distinct scopes from our work.

Testing other LVMs. Other LVMs such as JavaScript engines are heavily tested for quality assurance via generative [2, 17, 19, 21, 37, 44, 54, 60] or mutational [2, 20, 45, 55] fuzzing techniques and deep learning techniques [29, 60]. There has been work on testing other LVMs such as BPF [38, 39, 57], Ethereum [16], and Pharo LVM [47]. Among them, the most related are JIT-Picker [3], FuzzJIT [56], and Jitterbug [39]. JIT-Picker uncovers JIT-compiler bugs of JavaScript engines by differentially testing their interpreter and JIT compiler's fine-grained internal state (i.e., intermediate values of variables at specific program points) like the traditional approach. FuzzJIT wraps existing code with a loop template to trigger JIT compilation. Albeit similar to `LI`, FuzzJIT is specific to the loop template and unaware of the existence of the large compilation space. Jitterbug applies formal methods to model JIT correctness and verify BPF JITs. However, they target JITs implemented as AOT compilers in a restricted environment like the Linux kernel. All efforts on LVM testing have found many bugs in popular LVMs such as V8 and BPF. It would be promising to extend `Artemis` to other LVMs like JavaScript engines and BPF LVMs by leveraging CSE for validating the JIT compilers.

Testing compilers. More research has concentrated on compilers. Program generators like Csmith [59] and YARPGen [32, 33] can produce random C programs. SPE applies skeletal program enumeration to generate C programs [62]. Alive [35] and Alive2 [34] attempt to validate optimizations.

Table 5. 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-Compiler Specific": 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.

Related JVM Testing Tools	Venue	# Reported Bugs	Test Generation	Test Input Format	Test Method	Syntactical-Valid	Semantic-Preserving	JIT-Compiler Specific	Space Exploration	To what extend can generated tests finally reach the JIT compiler?
Sirer et al. [49]	DSL '99	–	G	B	D	✓	–	×	×	Occasionally reach
Yoshikawa et al. [61]	QSIC '03	–	G	B	D	✓	–	✓	×	Relies on AOT compilation
JavaFuzzer [18]	–	–	G	S	D	✓	–	×	×	Occasionally reach
JFuzz [1]	–	–	G	S	D	✓	–	×	×	Occasionally reach
dexfuzz [26]	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 [64]	ICSE '22	10	M	B	D	✓	×	×	×	Depends on ingredients
JAttack [63]	ASE '22	6	G	S	D	✓	–	×	×	Depends on templates
JITfuzz [58]	ICSE '23	36	M	S	D	✓	×	✓	×	Depends on seeds and mutators
JOpFuzzer [24]	ICSE '23	41	M	S	P	✓	✓	✓	×	Depends on JVM options
`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, requiring no other LVMs.

Another popular technique is EMI (Equivalence Modulo Input), a practical, effective idea that validates compilers by a seed program and its EMI variants and has found thousands of bugs in GCC and LLVM [27]. Practical tools based on EMI exploit dead or live code regions to derive EMI variants. Specifically, Orion randomly prunes the dead code from a seed program [27]; Athena enforces Markov Chain Monte Carlo (MCMC) to guide dead code deletion [28]; and Hermes inserts semantics-preserving code into the live area [51].

Conceptually, JoNM belongs to EMI family, especially live-code mutation. However, our mutations are fundamentally different from all proposed EMI work. First, JoNM aims to approximate CSE whose ultimate goal is to exhaustively explore the compilation space modulo the LVM under validation. 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.

6 Conclusion

We have presented the novel concept of compilation space modulo LVM and an effective approach CSE for JIT-compiler validation. We proposed JoNM to approximate CSE, 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 believe that the generality of CSE and JoNM likely makes them applicable and effective in other LVMs such as validating the JIT compilers of JavaScript engines. This work introduces and opens this promising line of exploration.

Acknowledgments

We are grateful to the anonymous reviewers and our shepherd, Shan Lu, for their valuable feedback on earlier versions of this paper. We are also indebted to the respective JVM developers for inspecting and fixing our reported bugs. This work was supported in part by the National Natural Science Foundation of China under Grants No. 61932021, No. 62272218, the Leading-edge Technology Program of Jiangsu Natural Science Foundation under Grant No. BK20202001, and the Fundamental Research Funds for the Central Universities (020214380102, 020214912220). The authors would like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. Yanyan Jiang (jyy@nju.edu.cn) is the corresponding author.

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] 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).*

[16] 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).*

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

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

[19] 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).*

[20] 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).*

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

[22] 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).*

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

[24] 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).*

[25] 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).*

[26] 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).*

[27] 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).*

[28] 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).*

[29] 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 (Security '20).*

[30] 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).*

[31] 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).*

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

[33] 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).

[34] 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)*.

[35] 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)*.

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

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

[38] 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)*.

[39] 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)*.

[40] 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)*.

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

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

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

[44] 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)*.

[45] 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)*.

[46] 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).

[47] 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)*.

[48] 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)*.

[49] 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)*.

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

[51] 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)*.

[52] 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)*.

[53] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. 2022. Mem-Liner: 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)*.

[54] 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)*.

[55] 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)*.

[56] 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 (Security '23)*.

[57] 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)*.

[58] 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)*.

[59] 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)*.

[60] 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)*.

[61] 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)*.

[62] 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)*.

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

[64] 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)*.