Validating JIT Compilers via Compilation Space Exploration

Anonymous Author(s)

Abstract

We introduce *compilation space exploration* (CSE), a novel, widely-applicable method for testing the just-in-time (JIT) compilers of modern language virtual machines (VMs). Our key insight is to systematically explore the large compilation space of a JIT compiler and differentially test the resulting JIT compilations of a program on a single VM. To create a lightweight, VM-agnostic realization of CSE, we strategically mutate test programs by leveraging JIT-relevant operations (e.g., loops). We realize our technique in `Artemis`, a tool for the Java virtual machine (JVM). Our evaluation using `Artemis` has led to 85 bug reports for three widely-used production JVMs, namely HotSpot, OpenJ9, and the Android Runtime, of which 53 have already been confirmed or fixed. All the reported bugs concern JIT compilers, and many are critical, 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; this work opens this promising direction.

Keywords: JIT compiler, JVM, virtual machines, testing

1 Introduction

Modern language virtual machines (VMs) such as the Java virtual machine (JVM) and JavaScript engines are among the most critical, widely-used system software ever written. In VMs, a source program is commonly run either by interpreting bytecode or by executing machine code, which is dynamically compiled by just-in-time (JIT) compilers. Well-known JIT compilers include HotSpot C1/C2, Graal, V8 Turbofan, and Berkeley Packet Filter (BPF) JIT in the Linux kernel. The JIT compiler, usually an optimizing compiler, is one of the most complicated components in modern VMs. It improves the runtime performance by exploiting a variety of analysis and optimization techniques like global value numbering, loop unrolling and vectorization, all of which are considered intricate and error-prone [64]. This makes the JIT compiler a primary source of bugs for modern VMs [43, 53, 61, 63].

In this paper, we refer to *JIT compiler bugs* as bugs that do *not* manifest when JIT compilers are disabled (e.g., by the `-Xint` option in HotSpot). Like other compiler bugs, for instance C [29, 64], JIT compiler bugs can be generally classified into two categories, *crash*es and *mis-compilation*s, where the first causes the VM to crash during the compilation or execution of the compiled machine code, and the second compiles bytecode to incorrect machine code.

Recently, both industry and academia have invested significant effort in testing VMs and achieved promising results [11, 12, 22, 41, 43, 48, 49, 68, 69]. However, due to their inherent limitations, we still lack effective techniques to specifically test JIT compilers. First, JIT compiler bugs are typically deep while tests generated by existing techniques are not designed to capture deep bugs—a large number of the generated tests are syntactically invalid [12, 22, 41] or semantically invalid [11, 69], leading to few tests capable of exercising the deep JIT compiler. On the other hand, even for tests that are able to exercise the JIT compiler, it cannot be systematically tested since these tests fail to explore the immense compilation space [43, 48, 49, 68]. Consequently, most of the disclosed bugs are shallow and irrelevant to JIT compilers, for example, early-stage parser or verifier bugs.

One mitigation to this problem might be the KEX[1] approach which involves executing a program twice: one with the JIT compiler forced to compile every method right before invoking them for the first time (e.g., by `-Xcomp` in HotSpot), and the other with the JIT compiler completely disabled (e.g., by `-Xint` in HotSpot) [28, 61]. This approach then compares the final program outputs and reports a JIT compiler bug if they differ. Due to its simplicity, KEX has been widely used in both industry [1, 41] and academia [4]. However, as we will discuss next, it is unable to explore the large compilation space even with respect to a single program. Thus, it has limited bug-finding capability.

Compilation space modulo VM. This paper introduces *compilation space modulo VM* (compilation space for short), a simple yet novel concept for testing VMs, especially their JIT compilers. Assuming a simplified VM that does not support on-stack-replacement [16], de-optimization [23], and background compilation, the compilation space of a program with n method calls consists of $O(2^n)$ likely JIT compilations because, for each method call, the VM can either interpret the method or opt for the JIT compiler and execute the machine code. The key of the compilation space is that the program outputs from running any JIT compilation are the same. It should be noted that the compilation space is considered *far more complicated* (i.e., containing far more JIT compilations) in terms of a full-featured VM; we formalize this in Section 3.

In contrast, the KEX approach can only consider few compilations, rather than the huge compilation space. Figure 1 depicts the compilation space of a simple Java program, assuming a simplified JVM as aforementioned. The program

[1] KEX is short for "constant times of execution" because some work executes for a third time with all the default VM options.

```
class T {
  void baz() {}
  void bar() { baz(); }
  void foo() { bar(); }
  public static void main(
    String[] args) {
    foo();
  }
}
```

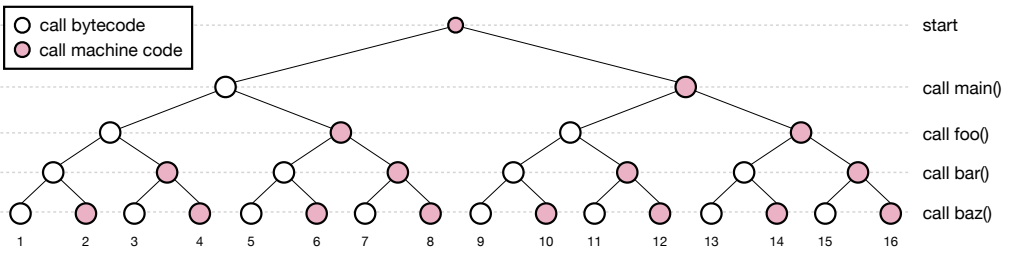

Figure 1. The compilation space (right) of a simple Java program (left), assuming a simplified JVM which does not support on-stack-replacement, de-optimization, and background compilation. For each method call, the JVM can either interpret the method or opt for the JIT compiler and execute the machine code after compilation.

consists of 4 method calls and thereby the space produces 16 likely compilations. KEX is capable of generating the left-most (the 1st) and right-most (the 16th) compilations while it cannot generate any of the others in between.

Compilation space exploration. This paper further proposes to systematically explore the compilation space. The basic idea is to guide the VM and the JIT compiler under testing to compile different parts of a single program in order to progressively explore all likely JIT compilations. Since the program output from running each JIT compilation should be the same, a JIT compiler bug will be reported if there are any output deviations between any two JIT compilations. Our high-level idea is to take a real-world program, execute it for a sufficient number of times where each time we drive the JIT compiler to partially compile a distinct code segment, and ensure the program outputs are the same. We name this *Compilation Space Exploration* (CSE).

An ideal realization of CSE is to feature a VM under testing with the ability for on-demand JIT compilation, but this requires substantial VM-specific manual effort. Instead, this paper proposes "JIT-Op Neutral Mutation" (JoNM), a novel, systematic strategy which can control the VM to make different decisions on when and how to JIT-compile which code segment of a program, through simple source-level program mutations rather than cumbersome VM-level VM modifications. The crux is to leverage JIT-relevant operations (JIT-ops) such as loops and method calls during program mutation. In particular, given a language virtual machine `VM` and a seed program P, JoNM derives a subset $\mathcal{P}$ of P's mutants by stochastically sampling a corpus of P's methods to insert, delete, or modify JIT-ops. JoNM guarantees all mutants $P' \in \mathcal{P}$ to be semantically equivalent to the seed program P, and pinpoints a JIT compiler bug if one of their program outputs diverges: $\exists P' \in \mathcal{P}.\texttt{VM}(P) \neq \texttt{VM}(P')$.[2] Through JoNM, we are able to steer the VM to deviate from one JIT compilation to another and jump among different JIT compilations in the space with flexibility, for instance, jumping from the 1st to the 6th in Figure 1. Furthermore, considering JIT-ops are generally similar among all VMs, a JoNM implementation is applicable to all implementations of a single VM (e.g., HotSpot and OpenJ9 of JVM, V8 and SpiderMonkey of JavaScript engines).

We have implemented JoNM for JVM in the `Artemis` tool. It focuses on program mutations via two kinds of JIT-ops: method calls and loops. Our evaluation on top of three widely-used production JVMs, namely HotSpot, OpenJ9, and the Android Runtime (ART), clearly shows `Artemis`' effectiveness. Specifically, we have filed a total of 85 bugs for three tested JVMs, where 53 have been confirmed or fixed as of 10 April 2023. It is worth mentioning that all our reported bugs are JIT compiler bugs and many are critical: 12 OpenJ9 bugs are tagged as `blocker`, the most severe, release-blocking type of bugs; 10 HotSpot bugs are marked as at least `P3`, i.e., major loss of function; and 13 OpenJ9 bugs are long latent across ≥4 major and many minor releases. Furthermore, our reported bugs resulted from diverse errors in various JIT compiler components (e.g., loop optimization and code generation). We also received positive feedback from both HotSpot and OpenJ9's 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."*

Contributions. Overall, our contributions are as follows:

- We introduce the novel concept of Compilation Space modulo VM, with the key that the program outputs from running all JIT compilations within the space are the same.
- We propose Compilation Space Exploration, a simple, widely-applicable methodology for testing JIT compilers, aiming to explore the compilation space systematically and discover JIT compiler bugs through JIT compilation's differential testing.
- We propose the JIT-Op Neutral Mutation strategy to simulate CSE from the source-code level, a novel VM-agnostic realization for CSE.
- We implement JoNM for JVM as the `Artemis` tool and conduct an extensive evaluation that results in 85 bugs

[2] Without loss of generality, this paper assumes that every test program requires no program inputs. Therefore, we omit the input argument I of the formula $\texttt{VM}(P, I)$ throughout the whole paper.

for 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://anonymous.com/anonymous/anonymous

2 Illustrative Examples

This section presents the required background on JIT compilers and a concrete HotSpot bug to motivate and illustrate CSE as well as `Artemis`. The bug originates from a JavaFuzzer-generated test [19] and manifests by `Artemis`' mutations.

2.1 Background: JIT Compilers

JIT compilers are essential components of numerous critical system software like modern VMs, emulators [3], and specialization [26, 39]. They aim to improve the runtime performance by dynamic compilation and native execution, while also providing fast start-up times through interpretation [9, 13, 27, 40]. To this end, the JIT compiler translates bytecode into machine code on-the-fly, and the VM then executes the compiled machine code directly on the underlying hardware, rather than interpreting the bytecode. Nevertheless, the JIT compilation is not immediately activated upon VM start-up. Instead, the VM profiles bytecode when interpreting to spot and compile frequently interpreted code segments, or "*hot code*". In particular, the VM maintains a counter for each method and control-flow back-edge (i.e., a loop). It increments the counter whenever the corresponding method is invoked or back-edge is executed. When a counter reaches a predefined compilation threshold, the associated hot code (method or loop) is queued for JIT compilation, where the JIT compilation of hot loops is also known as *OSR (On-Stack Replacement) compilation* since continuing the execution of a method from the middle involves replacing the active bytecode stack frame by a native stack frame [16].

Usually, the JIT compilation is leveled (or tiered) and proceeds progressively [24, 46]. That said, the hot code is likely re-compiled or further optimized at a higher level as the counter gets larger. Higher-level compilations are expected to be more aggressive and produce more efficient machine code at the cost of more compilation time.

In addition to compilation and optimization, there are chances that the VM bails out to the interpreter from the compiled machine code. This is because the JIT compilation is often based on profiled, compile-time speculative, and optimistic assumptions; any violation against the assumptions invalidates the compiled machine code, forcing the VM to continue by re-interpreting bytecode. This process is typically termed *de-optimization* [23] and the circumstances that can lead to de-optimization are called *uncommon traps*. For example, JVM often de-virtualizes an `invokevirtual` call to an `invokedirect` call when CHA (Class Hierarchy Analysis) infers that there is exactly one implementation for the invoked method inside JVM. However, JVM has to de-optimize when this assumption is broken, i.e., a new implementation (e.g., a subclass) for the invoked method is loaded. Other uncommon traps include violating the assumptions of null-check, bounds-check, etc. De-optimization causes the counter to be decremented.

```
class T {
  boolean z = false; byte l = 0;
  void g() {
    // (some omitted lines...)
    for (int m : k) {
      // (some omitted lines...)
      switch ((m >>> 1) % 10 + 36) {
      case 36:
        for (int w = -2967; w < 4342; w += 4);
        new PrintStream(new OutputStream() {
          public void write(int b) {}
        });
        l += 2;
      case 40: break;
      case 41: k[1] = 9;
      }
    }
  }
  void o() { if (z) { return; } g(); }
  void p() {
    for (int q = 2; q < 5; ++q) {
      z = true;
      for (int u = 0; u < 9676; u++)
        o();
      z = false;
      o();
    }
    System.out.println(l);
  }
  public static void main(String[] q) {
    T t = new T(); t.p(); t.p();
  }
}
```

Figure 2. JDK-***** triggers a mis-compilation in HotSpot. JavaFuzzer generates the seed while `Artemis` inserts the highlighted code snippets. Code shown in this example is a cleaned-up version from a very large test program.

Note that different VMs employ different static/dynamic compilation thresholds/heuristics to increment/decrement counters. This section provides only a high-level overview.

2.2 Illustrative HotSpot Example

`Artemis` detects JIT compiler bugs in the JVM by applying JoNM (Section 1 and Section 3.3) to Java source programs which we hereafter call seeds. Specifically, `Artemis` mutates the seed program by synthesizing loops and inserting them into the seed program. The synthesized loops are guaranteed neutral so that they will not affect the semantics of the seed program. As a result, `Artemis`' mutations guide the JVM to produce different JIT compilations when running the mutants. By comparing the program outputs between a seed program and its mutants, a JIT compiler bug can be spotted if the outputs are different.

HotSpot JDK-***.** Figure 2 presents a test that causes HotSpot to mis-compile. It was detected at OpenJDK 11.0.15 (revision `f915a327`) but also affects JDK 17 and 20. In this example, we used JavaFuzzer to generate the seed program and `Artemis` to derive the mutant (highlighted). Since the original test case is large and complex, we reduced it automatically using Perses [57] and C-Reduce [52], and performed further manual cleanup.

The seed keeps incrementing `T.l` by 2 (Line 13) in a loop (Line 5) and printing its value (Line 28). In this example, all T's methods are interpreted until the seed exits since no compilation thresholds are reached. Upon receiving the seed, `Artemis` attempts two kinds of mutations. First, it tries to JIT-compile `T.o()` by pre-invoking it for `9676` times (Lines 23-24) before the actual method call at Line 26. Given that directly invoking `T.o()` interferes with the semantics (i.e., resulting in different `T.l`), `Artemis` inserts an additional control flag `z` and a piece of control prologue into `T.o()` (highlighted, Line 19) so that it can return early while being pre-invoked. Second, `Artemis` heats up `T.g()` by introducing a simple loop at Line 9.[3] Such mutations, albeit simple, notably influence how HotSpot executes the program:

- `T.o()` is first JIT-compiled by C1 at `L3` level and then JIT-compiled again by C2 at `L4` level once it is hot enough. After that, it is de-optimized when called at Line 26 because the JIT compiler assumes `z == true`.
- The loop at Line 9 is OSR-compiled by C2 at `L4` level supposing that `w < 4342` and de-optimized when the loop exits. `T.g()` is also JIT-compiled at `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 pass incorrectly moves a store (to `T.l`) instruction 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 adjusted their frequency estimation heuristics to prevent such illegal movements.

It is worth mentioning that this bug cannot be detected simply by the KEX approach, as it requires the JIT compiler to (1) partially compile some specific code segments and (2) de-optimize. We provided another OpenJ9 bug in our supplementary material to illustrate CSE and `Artemis`.

3 CSE and The `Artemis` Implementation

This section formalizes compilation space modulo VM (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 VM

A VM typically maintains a set of $M+1$ counters $C_m = \{c_i \mid 0 \le i \le M \wedge c_i \ge 0\}$ for a given method m with M back-edges. These counters include the method counter, denoted by c_0, and the back-edge counters, denoted by c_1 through c_M. To facilitate multi-level compilation, a VM usually defines N compilation thresholds $Z_1, \cdots, Z_N$, where $0 \le Z_i < Z_{i+1} < +\infty$. These compilation thresholds divide the counter values into $N+1$ ranges: $[Z_{i-1}, Z_i)$; without loss of generality, this paper sets $Z_0 = 0$ and $Z_{N+1} = +\infty$.

This paper measures the hotness of a counter and a method by *temperature*. Specifically, a counter c is said to have temperature $\tau(c) = \mathtt{t}_i$ if and only if $c \in [Z_i, Z_{i+1}) \wedge 0 \le i \le N$, where the temperature $\tau(c)$ satisfies a total order, i.e., $\mathtt{t}_i < \mathtt{t}_{i+1}$ always holds for $0 \le i \le N-1$.

Definition 3.1 (Temperature)**.** A method m's temperature, $\tau(m)$, is determined by the maximal temperature of all its counters C_m:

$$\tau(m) = \max_{c \in C_m} \tau(c).$$

A method with temperature $\mathtt{t}_0$ indicates that it is being interpreted, while a method with temperature $\mathtt{t}_{i\neq 0}$ implies that it is executing machine code, i.e., it has already[4] been JIT- or OSR-compiled at the i-th compilation level.

A called method can be heated up by method calls and loops, and cooled down by colorful uncommon traps; this paper names them *JIT-relevant operations* (JIT-ops). The *temperature vector* u^i_m of method m tells how its temperature changes over time when it is called at the i-th time. The temperature change is typically induced by various JIT-ops. The temperature vector reflects how a VM compiles and de-optimizes m when it is invoked for the i-th time. For example, we can infer from $u^i_m = \langle \mathtt{t}_0, \mathtt{t}_1, \mathtt{t}_0 \rangle^i_m$ that: m is interpreted when it is immediately called for the i-th time, but it is subsequently heated up and compiled at level 1 through JIT or OSR compilation; however, it is then de-optimized and re-interpreted until this method call is completed.

A *JIT compilation* (JIT-comp) is a sequence of temperature vectors. It is analogous to an annotated method call trace, reflecting how a VM executes (interprets or compiles) a program *method call by method call*. Note that every program comes with a default JIT-comp for every VM; this is the one generated when running the program with all default JIT compiler-related options. The following shows an example JIT-comp of a program with respect to OpenJ9:

$$\begin{aligned}\varphi = \langle \mathtt{t}_0 \rangle^0_{\mathtt{T.main}} \to \langle \mathtt{t}_0 \rangle^0_{\mathtt{T.T}} \to \langle \mathtt{t}_0 \rangle^0_{\mathtt{T.f}} \to \langle \mathtt{t}_0 \rangle^0_{\mathtt{T.b}} \to \\ \cdots \to \langle \mathtt{t}_0 \rangle^k_{\mathtt{T.f}} \to \langle \mathtt{t}_2 \rangle^k_{\mathtt{T.b}} \to \\ \cdots \to \langle \mathtt{t}_0 \rangle^{211}_{\mathtt{T.f}} \to \langle \mathtt{t}_2 \rangle^{211}_{\mathtt{T.b}} \to \langle \mathtt{t}_0 \rangle^0_{\mathtt{System.out.println}}.\end{aligned}$$

It tells that (1) the first k calls to `T.b()` enable itself to be JIT-compiled at temperature $\mathtt{t}_2$, the `warm` compilation level in OpenJ9, (2) all subsequent calls to `T.b()` directly execute the compiled machine code, and (3) all other methods are continuously interpreted until `T.main()` exits.

[3] Lines 10–12 are part of our synthesized code that aims to make the mutation neutral. Section 3.4 contains further details.

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

Considering that a JIT compiler can compile, optimize, and de-optimize a method at many theoretically[5] valid program points beyond the method entry, executing a single program with n method calls can generate $\Omega(2^n)$ likely JIT-comps with respect to a single VM.

Definition 3.2 (Compilation Space modulo VM)**.** Given a program P and a language virtual machine VM, all JIT-comps that can be generated by VM with respect to program P constructs the compilation space of P modulo language virtual machine VM:

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

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

3.2 Compilation Space Exploration

The crux of the compilation space concept is that a VM should always yield *the same program output* no matter which JIT-comp $\varphi \in \mathbb{S}_{\texttt{VM}}(P)$ is generated when executing program P. Therefore, a JIT compiler bug exists in VM when

$$\exists\varphi_1\exists\varphi_2.\, \varphi_1 \in \mathbb{S}_{\texttt{VM}}(P) \wedge \varphi_2 \in \mathbb{S}_{\texttt{VM}}(P) \longrightarrow \texttt{VM}(P, \varphi_1) \neq \texttt{VM}(P, \varphi_2).$$

Our goal is to systematically explore the compilation space of a real-world program and pinpoint JIT compiler bugs of a VM by validating the program output of *every* legitimate JIT-comps in the space. We name this *Compilation Space Exploration* (CSE).

Possible realizations. An ideal realization of this goal is to equip a VM with the ability of on-demand JIT compilation. However, this requires considerable engineering effort and is considered infeasible in practice because (1) practical VMs are specially designed to allow JIT/OSR compilation and de-optimization only at specific program points, which makes some theoretically valid JIT-comps invalid, and (2) the space for even a small program is vast, often difficult or even impossible to compute due to implicit builtin library method calls (e.g., a single `System.out.println()` call involves dozens of builtin method calls). Another issue with this realization is that the implementations are tightly bound to specific VMs, thus not portable.

A practical realization is to fuzz the JIT compiler-related options of a VM like JOpFuzzer [25], but (1) this needs substantial expertise and manual work to understand every JIT compiler option in order to generate valid JIT-comps, and (2) the space exploration capability is largely constrained by the number and effects of VM options. In addition, the understanding of one VM's options cannot be used by other VMs, 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 VM options. These motivated us to look for a new CSE realization.

[5]Some program points are considered valid for JIT/OSR compilation and de-optimization, while may be forbidden by a practical VM for some specific considerations like performance.

3.3 JIT-Op Neutral Mutation

In this paper, we present a novel strategy called "JIT-Op Neutral Mutation" (JoNM) which attempts to simulate CSE from the source-code level with the help of JIT-ops, while being lightweight, VM-agnostic, and practical for any VM (JVM, JavaScript engines, etc.).

We leverage the feature that VMs intensively rely on JIT-ops for JIT compilation and de-optimization. Specifically, VMs require method calls and loops to enable JIT compilation and uncommon traps to de-optimize. Our insight is that controlling the use of JIT-ops can help guide the VM and the JIT compiler under testing to JIT/OSR-compile or de-optimize distinct code segments of a single seed program. We rely on the mutations to steer a VM to deviate from one JIT-comp to another and explore the whole compilation space progressively as more mutants are generated.

In particular, given a seed program P, JoNM derives a subset $\mathcal{P}$ of P's mutants by stochastically sampling a corpus of P's methods to insert, delete, or modify JIT-ops (i.e., method calls, loops, and uncommon traps), while guaranteeing the mutations *neutral* to P's semantics. That said, every generated mutant $P' \in \mathcal{P}$ is expected to (1) produce a different JIT-comp from P (by JIT/OSR-compiling a distinct code segment or de-optimizing at a distinct program point), and (2) preserve the same program output as P's. In this way, the systematic exploration of P's compilation space modulo VM can be simulated by running a sufficient number of P's mutants. Hence, a JIT compiler bug exists in VM if

$$\exists P'.\, P' \in \mathcal{P} \longrightarrow \texttt{VM}(P) \neq \texttt{VM}(P').$$

For simplicity, we omit the JIT-comp argument and directly use $\texttt{VM}(P)$ when executing P by requiring VM to generate the default JIT-comp.

Versus other realizations, JoNM has several advantages:

- *Lightweight and simple*: JoNM simulates CSE at the source level, thus requiring negligible manual effort to understand VMs and no modifications to the VMs.
- *VM-agnostic and widely-applicable*: Since JIT-ops are typically similar among all implementations of the same type of VMs (e.g., HotSpot and OpenJ9 for JVM, V8 and SpiderMonkey for JavaScript engines), a single JoNM implementation can be used to test the same types of VM implementations (e.g., all JVM implementations).
- *Practical*: Since JoNM can generate tests based on either real-world programs or 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 testing JIT compilers.

Algorithm 1: The `Artemis` procedure for testing JIT

```
1  procedure Test(VirtualMachine VM, Program P)
2      R ← VM(P)                    // Run P with P's default JIT-comp
3      for i ← 1...MAX_ITER do
4          P' ← JoNM(P)
5          R' ← VM(P')              // Run P' with P''s default JIT-comp
6          if R' ≠ R then           // Discrepancies imply bugs
7              ReportJITCompilerBug(P')
8  function JoNM(Program P)
9      P' ← P
10     foreach Method m ∈ P'.Methods() do
11         if FlipCoin() then
12             φ ← Random mutator from LI, SW, and MI
13             ρ ← Random program point within method m
14             L ← SynLoop(φ, ρ)
15             P' ← φ.Mutate(P', m, ρ, L)
16     return P'
```

3.4 The `Artemis` Implementation

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

Algorithm 1 describes `Artemis`' main process. For each seed P, `Artemis` attempts to mutate it (Line 4) and run the mutant P' with its default JIT-comp (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 overriden 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), with the help of a synthesized loop L which could heat up m to a higher temperature at program point ρ (Line 14). Finally, the synthesized loop L would be 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., synthesizing programs by filling holes left in a predefined skeleton [55, 67]. 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 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 backups 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'.

Algorithm 2: Loop synthesis for JoNM

```
1  function SynLoop(Mutator φ, ProgPoint ρ)
2      L ← φ.loop_skeleton          // Initialized as the skeleton
3      V ← ρ.Variables()            // Variable set available at ρ
4      V' ← ∅                       // Saving reused variables in synthesis
5      foreach ExprHole ℏ ∈ L.expr_holes do
6          L ← Substitute(L, ℏ, SynExpr(ℏ, V, V'))
7      foreach StmtsHole ℏ ∈ L.stmts_holes do
8          L ← Substitute(L, ℏ, SynStmts(ℏ, V, V'))
9      foreach Variable v ∈ V' do
10         L ← Backup v; L; Restore v;
11     return L
12 function SynExpr(ExprHole ℏ, VarSet V, VarSet V')
13     T = GetType(ℏ)
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 ∈ V with the
              type T; meanwhile expand V' by V' ← {v} ∪ 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 ℏ, VarSet V, VarSet V')
21     S ← Random statement skeleton
22     foreach ExprHole ℏ ∈ S.expr_holes do
23         S ← Substitute(S, ℏ, SynExpr(ℏ, V, V'))
24     return S
```

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

- For primitive-like types including boxed and unboxed [47] 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 component type T.`comp_type`, dimension T.`dimen`, and 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

```
1  for (int i=min(MIN,<expr>); i<max(MAX,<expr>); i+=STEP) {
2    <stmts>;
3  }                                        // LI.loop_skeleton
4  ----------------------------------------------------------
5  boolean exec = false;
6  for (int i=min(MIN,<expr>); i<max(MAX,<expr>); i+=STEP) {
7    <stmts>;
8    if (!exec) { <placeholder:stmt>; exec = true; }
9    <stmts>;
10 }                                        // SW.loop_skeleton
11 ----------------------------------------------------------
12 for (int i=min(MIN,<expr>); i<max(MAX,<expr>); i+=STEP) {
13   <stmts>;
14   P.m_ctrl = true; <placeholder:method>; P.m_ctrl = false;
15   <stmts>;
16 }                                        // 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.

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 VM testing [20]. 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 tested 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 affected. 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).

Method Invocator. MI is designed to trigger JIT compilation in addition to OSR compilation. 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 calling 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 22–25 is our synthesized loop L which pre-calls T.o() for 9676 times; and the method call o() at Line 26 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-calling T.o(). During pre-calls, our synthesized code highlighted at Line 19 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 conflict, (2) replace System.out and System.err by a PrintStream that prints nothing (e.g., Figure 2, Lines 10–12) 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 [50] to parse the Java source code as well as for enabling 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 VM testing [20].

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 tested JVMs with at least 16 bugs for each VM.
- **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 high quantity and quality of our reported bugs have demonstrated the clear effectiveness of our approach in finding JIT compiler bugs, and (2) at least 16 bugs per JVM shows the general applicability of our approach.

4.1 Testing 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 [11, 12, 69]. ART was selected as our subject because of its tremendous user base [14]. 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 to test because they are long-term supported (LTS). ART is excluded from choosing JDK versions because it does not support class bytecode directly.[6] For each selected JVM, we built its latest trunk version at the time of testing, and tested it with (1) background compilation (if supported) disabled and (2) 1GiB Java heap memory. We did not test 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 hinder us from distinguishing whether a newly detected bug duplicates an existing one. Finally, our testing mainly focused on the x86_64 Linux platform.

Seed programs. We used JavaFuzzer [19], 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 [57] and C-Reduce [52]. 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.

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

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

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; thereby in our evaluation, we set `MAX_ITER` to 8 to simulate exploring eight JIT-comps 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 tested JVMs, including 32 in HotSpot, 37 in OpenJ9, and 16 in ART. The first half of Table 1 presents the current status of the reported cases. As of 10 April 2023, 53 of them have already been confirmed and 26 have been fixed. We recognize a reported bug as "Confirmed" if the corresponding VM developers can reproduce the bug in their settings. Otherwise, we leave them in the "Reported" category regardless 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 categorized 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 machine 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 machine code. The symptoms are various, e.g., segmentation faults or assertion failures.

Performance issue. Executing the compiled machine code causes JVM to be obviously slower than interperting the bytecode. This is typically user-perceivable and there are chances that the JVM process is finally killed by the underlying 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 [4]. 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 bugs; 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 our testing effort and even asked "*Do you think there are going to be many more?*"

Furthermore, we received very positive feedback from the respective VM 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.

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 machine 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 [45]	1
Code Generation, C2	3	Other JIT Compone.	6
Code Execution, C2	3	Garbage Collection	13

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.

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 [15], suggesting that JIT compiler bugs pose a significant threat as they can impact various VM 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 a single 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 [50] are booted only once but drived

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

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 KEX to show the power of exploring more JIT-comps. We also collected relevant statistics and measured the throughput of `Artemis`.

In this study, we reused the synthesis parameters mentioned in Section 4.1 and chose OpenJ9 (JDK 11, revision `4ca209b5`) as the test target. We used JavaFuzzer as the seed generator. For each seed, we first ran it once with its default JIT-comp in OpenJ9. Next, we ran it again by forcing every method to be JIT-compiled before their first invocations by the `-Xjit:count=0` OpenJ9 option. In this section, we call the JIT-comp generated in this manner "forced-JIT JIT-comp". Then, we mutated the seed 8 times using `Artemis` and ran each mutant with their default JIT-comps. 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 cannot finish within 2 minutes. Table 4 presents the results.

Results. During 7 days, `Artemis` drived JavaFuzzer to generate 42,559 seeds and mutated them for 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 JIT-comp with the forced-JIT one. 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 builtin method calls, which is beyond the current capability of `Artemis`. 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.

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

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

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. We have discussed a small selection to highlight the diversity in our supplementary material. Readers are also encouraged to visit `Artemis`' website for a complete reference to each reported JIT compiler bug.

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 [4]—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 VM. 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 as interesting and promising future work.

Capabilities and limitations. In theory, CSE aims to systematically explore the compilation space of every real-world program and validate their program outputs. In practice, JoNM simulates the systematic space exploration by taking into consideration the portability issue 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 tested 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 builtin 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 testing. However, we decide 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 expected to be addressed with finer-grained approaches [36].

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 VMs as possible. Second, JoNM applies a 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 testing process. Third, integrating white-box techniques and expressive loop idioms [36, 63] into loop synthesis is also promising to explore. Finally, it would be interesting to extend our work to validate other VMs 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 testing lies at the intersection of VM testing and compiler testing. Thanks to the importance of VMs and compilers, both industry and academia have invested substantial effort to improve their quality. This section surveys the most relevant related work.

Testing JVMs. JVM testing is the most relevant thread of work; Table 5 shows a summary.

Sirer et al. proposed a program generator for Java bytecode following a production grammar [54]; JavaFuzzer [19] 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 [28]; classfuzz leverages code coverage to guide bytecode mutation and generation [12]; 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 [11]; JavaTailor extracts five types of code ingredients from historical bug-revealing programs and synthesizes mutants by inserting them to seed programs [69]; JAttack derives new tests by executing human-written skeletons and dynamically filling skeleton holes [68]. 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 [10] approach: CSE explores the whole compilation space and differentially tests any two JIT-comps of a single program and a single VM. Second, JoNM simulates this by differentially testing a seed program and its mutant inside a single VM. 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 [66]. 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 [28]. These efforts belong to the KEX family which compares the results of only a constant number of classical JIT-comps. However, CSE aims to explore the whole compilation space systematically. JITfuzz fuzzed JIT compiler guided by coverage and optimization-activating mutators [63]. JOpFuzzer explored and tested JIT compiler-related options [25]. Versus `Artemis`, both tools require substantial expertise and human effort for understanding 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 are orthogonal to ours and are promising to be integrated into CSE.

Finally, work on other JVM aspects such as side channels of JIT compilation [5, 6], type systems [7, 8], garbage collections [44, 58], and JVM performance [33, 34] were also proposed recently. These have distinct scopes from our work.

Testing other VMs. Other VMs such as JavaScript engines are heavily tested for quality assurance via generative [2, 18, 20, 22, 41, 48, 59, 65] or mutational [2, 21, 49, 60] fuzzing techniques and deep learning techniques [31, 65]. There has been work on testing other VMs such as BPF [42, 43, 62],

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-Specific Testing": whether the work specifically aims at JIT compilers; and "Systematic Exploration": whether the work can systematically explore the compilation space modulo the VM under testing.

		Venue	# Reported Bugs	Test Generation	Test Input Format	Test Method	Syntactical-Valid	Semantic-Preserving	JIT-Specific Testing	Systematic Exploration	To what extend can generated tests finally reach the JIT compiler?
Related JVM Testing Tools	Sirer et al. [54]	DSL '99	–	G	B	D	✓	–	×	×	Occasionally reach
	Yoshikawa et al. [66]	QSIC '03	–	G	B	D	✓	–	✓	×	Relies on AOT compilation
	JavaFuzzer [19]	–	–	G	S	D	✓	–	×	×	Occasionally reach
	JFuzz [1]	–	–	G	S	D	✓	–	×	×	Occasionally reach
	dexfuzz [28]	VEE '15	–	G	B	D	✓	–	✓	×	Relies on AOT compilation
	classfuzz [12]	PLDI '16	62	M	B	D	×	×	×	×	Occasionally reach
	classming [11]	ICSE '19	14	M	B	D	×	×	×	×	Occasionally reach
	JavaTailor [69]	ICSE '22	10	M	B	D	✓	×	×	×	Depends on ingredients
	JAttack [68]	ASE '22	6	G	S	D	✓	–	×	×	Depends on templates
	JITfuzz [63]	ICSE '23	36	M	S	D	✓	×	✓	×	Depends on seeds and mutators
	JOpFuzzer [25]	ICSE '23	41	M	S	P	✓	✓	✓	×	Depends on VM options
	`Artemis`	–	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 VMs (or compilers), i.e., requiring other VMs as references;
P: metamorphic testing: on the single VM under testing, not requiring any other VM.

Ethereum [17], and Pharo VM [51]. Among them, the most related are JIT-Picker [4], FuzzJIT [61], and Jitterbug [43]. 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) using KEX. 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 static (AOT) compilers in a restricted environment like the Linux kernel. All efforts on VM testing have found many bugs in popular VMs such as V8 and BPF. It would be promising to extend `Artemis` to other VMs like JavaScript engines and BPF VMs by leveraging CSE for validating the JIT compilers.

Testing compilers. More research has concentrated on compilers. Program generators like Csmith [64] and YARPGen [35, 36] can produce random C programs. SPE applies skeletal program enumeration to generate C programs [67]. Alive [38] and Alive2 [37] 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 and has found thousands of bugs in GCC and LLVM [29]. 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 [29]; Athena enforces Markov Chain Monte Carlo (MCMC) to guide dead code deletion [30]; and Hermes inserts semantic-preserving code into the live area [56]. CLSmith adapts EMI to OpenCL [32].

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 CSE whose goal is to systematically explore the compilation space modulo the VM under testing. Second, our mutations are applied on VM's (optimizing) dynamic JIT compilers instead of static compilers, where the former heavily interacts with the corresponding VM 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 VM and an effective method CSE for detecting JIT compiler bugs in modern VMs. We proposed JoNM to simulate CSE, a lightweight, VM-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 make them applicable and effective in other VMs such as validating the JIT compilers of JavaScript engines. This work introduces and opens this promising line of exploration.

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] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC '05)*.

[4] 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)*.

[5] 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)*.

[6] 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)*.

[7] 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).

[8] 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)*.

[9] 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)*.

[10] 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).

[11] 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)*.

[12] 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)*.

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

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

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

[16] 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)*.

[17] 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)*.

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

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

[20] 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)*.

[21] 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)*.

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

[23] 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)*.

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

[25] 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)*.

[26] Konstantinos Kallas, Tammam Mustafa, Jan Bielak, Dimitris Karnikis, Thurston H.Y. Dang, Michael Greenberg, and Nikos Vasilakis. 2022. Practically Correct, Just-in-Time Shell Script Parallelization. In *Proceedings of the 2022 USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*.

[27] 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)*.

[28] 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)*.

[29] 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)*.

[30] 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)*.

[31] 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)*.

[32] 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)*.

[33] 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)*.

[34] 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)*.

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

[36] 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).

[37] 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)*.

[38] 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)*.

[39] Henry Massalin and Calton Pu. 1989. Threads and Input/Output in the Synthesis Kernal. In *Proceedings of the 1989 ACM Symposium on Operating Systems Principles (SOSP '89)*.

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

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

[42] 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)*.

[43] 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)*.

[44] 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)*.

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

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

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

[48] 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)*.

[49] 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)*.

[50] 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).

[51] 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)*.

[52] 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)*.

[53] Chris Rohlf and Yan Ivnitskiy. 2011. Attacking Clientside JIT compilers. *Black Hat USA* (2011).

[54] 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)*.

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

[56] 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)*.

[57] 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)*.

[58] 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)*.

[59] 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)*.

[60] 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)*.

[61] 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)*.

[62] 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)*.

[63] 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)*.

[64] 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)*.

[65] 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)*.

[66] 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)*.

[67] 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)*.

[68] 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)*.

[69] 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)*.