JITfuzz: Coverage-guided Fuzzing for JVM Just-in-Time Compilers

Mingyuan Wu†
Southern University of Science and Technology
Shenzhen, China and
The University of Hong Kong
Hong Kong, China
11849319@mail.sustech.edu.cn

Minghai Lu
Southern University of Science and Technology
Shenzhen, China
11910620@mail.sustech.edu.cn

Heming Cui
The University of Hong Kong
Hong Kong, China
heming@cs.hku.hk

Junjie Chen
Tianjin University
Tianjin, China
junjiechen@tju.edu.cn

Yuqun Zhang*
Southern University of
Science and Technology
Shenzhen, China
zhangyq@sustech.edu.cn

Lingming Zhang
University of Illinois Urbana-Champaign
Champaign, USA
lingming@illinois.edu

Abstract*—As a widely-used platform to support various Java-bytecode-based applications, Java Virtual Machine (JVM) incurs severe performance loss caused by its real-time program interpretation mechanism. To tackle this issue, the Just-in-Time compiler (JIT) has been widely adopted to strengthen the efficacy of JVM. Therefore, how to effectively and efficiently detect JIT bugs becomes critical to ensure the correctness of JVM. In this paper, we propose a coverage-guided fuzzing framework, namely *JITfuzz*, to automatically detect JIT bugs. In particular, *JITfuzz* adopts a set of optimization-activating mutators to trigger the usage of typical JIT optimizations, e.g., function inlining and simplification. Meanwhile, given JIT optimizations are closely coupled with program control flows, *JITfuzz* also adopts mutators to enrich the control flows of target programs. Moreover, *JITfuzz* also proposes a mutator scheduler which iteratively schedules mutators according to the coverage updates to maximize the code coverage of JIT. To evaluate the effectiveness of *JITfuzz*, we conduct a set of experiments based on a benchmark suite with 16 popular JVM-based projects from GitHub. The experimental results suggest that *JITfuzz* outperforms the state-of-the-art mutation-based and generation-based JVM fuzzers by 27.9% and 18.6% respectively in terms of edge coverage on average. Furthermore, *JITfuzz* also successfully detects 36 previously unknown bugs (including 23 JIT bugs) and 27 bugs (including 18 JIT bugs) have been confirmed by the developers.

I. Introduction

Java Virtual Machine (JVM) has been widely adopted in many popular application domains, e.g., mobile applications and cloud computing, by supporting the execution of Java bytecode compiled from various high-level programming languages, e.g., Java, Scala, and Clojure [1]. However, while JVM is advanced in adopting interpretation in addition to compilation for cross-platform execution, interpreting JVM-based programs incurs high performance overhead. To tackle this issue, the Just-in-Time compiler (JIT) has been designed to improve runtime compilation performance of JVM-based programs by compiling selected code (e.g., frequently executed code) into native machine code. In this way, the resulting JVM bytecode can be executed directly without the costly interpretation process, leading to efficient program execution. To date, JIT has become a crucial component in JVM implementations where its correctness plays a vital role in ensuring correct and efficient execution of JVM-based programs.

While it is evident that testing JITs to ensure their correctness is vital for the correct execution for JVMs, how to effectively and efficiently test JITs remains rather challenging due to the following reasons. First, JITs can hardly be thoroughly tested because they include diverse optimization techniques which are activated under diverse scenarios. That said, to thoroughly test JITs, it is essential to create as many such optimization scenarios as possible, which can be potentially challenging. Second, while random/probabilistic mutation becomes a major paradigm adopted by many fuzzers [2, 3, 4, 5], it is nevertheless inefficient for JIT testing since massive resulting mutants cannot conform to JVM specification and executing them easily terminates JIT testing early (e.g., in the input verification phase [6]) to prevent testing deep JIT states. At last, although traditional JVM fuzzing techniques have proven that applying control-flow mutators can improve testing effectiveness, such mutators can hardly be directly applied in fuzzing JITs, e.g., adding new transitions in the existing control flow graphs can easily break variable dependencies [7], causing early-terminated JVM executions and thus leading to insufficient JIT testing. Therefore, albeit general-purpose

† Mingyuan Wu is also affiliated with the Research Institute of Trustworthy Autonomous Systems, Shenzhen, China.

* Yuqun Zhang is the corresponding author. He is also affiliated with the Research Institute of Trustworthy Autonomous Systems, Shenzhen, China and Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation, China

JVM fuzzing techniques can occasionally expose JIT bugs, there still is a pressing need for a dedicated fuzzing technique specifically targeting JITs.

In this paper, we propose *JITfuzz*, a coverage-guided fuzzing framework specifically targeting JITs. In particular, testing JITs essentially is testing their mechanisms for compiling bytecode into native machine code during runtime. Intuitively, such mechanisms can expose *interesting* behaviors (which may expose bugs) when triggering the usage of their optimization techniques and enriching the control flows of target JITs. Therefore in this paper, after explicitly launching JITs via specific JVM commands, *JITfuzz* adopts two types of mutators to advance thorough testing of the JIT mechanism. More specifically, optimization-activating mutators are proposed to trigger the activation of typical optimization techniques, e.g., the function-inlining-activating mutator is applied to create scenarios where the function inlining strategy [8] is activated. Meanwhile, noticing that mutating program control flows can potentially affect JIT optimizations, we also propose control-flow-enriching mutators to *safely* complicate program control flows of the generated test programs (i.e., alleviating early termination). Furthermore, *JITfuzz* adopts a mutator scheduler to dynamically schedule the mutators to optimize the runtime testing coverage of JITs.

To evaluate the effectiveness of *JITfuzz*, we first construct a real-world benchmark suite composed of 6 commonly adopted projects from prior JVM fuzzing work [7, 9] and 10 popular open-source projects in GitHub [10]. Next, by choosing OpenJDK19 [11] as our target JVM, we conduct a set of experiments to explore the effectiveness of *JITfuzz* and its components. The evaluation results suggest that *JITfuzz* outperforms state-of-the-art mutation-based JVM fuzzer *Classming* [7] and generation-based JVM fuzzer *JavaTailor* [9] by 27.9% and 18.6% respectively in terms of the code coverage. Meanwhile, all our proposed technical components in *JITfuzz*, including the mutators and mutator scheduler, are effective. For instance, adopting mutator scheduler can increase the code coverage by 10.2% on average. Moreover, *JITfuzz* successfully detects 36 previously unknown bugs on three commercial JVMs while none of them can be detected by *Classming* or *JavaTailor*. Specifically, 27 bugs have been confirmed by the corresponding developers and 16 have already been fixed. Moreover, 23 of them are JIT bugs where 18 have been confirmed and 7 have been fixed.

In summary, our paper makes the following contributions:

- **Approach.** To our best knowledge, we propose the first coverage-guided fuzzing framework for JVM JIT namely *JITfuzz* with specifically designed mutators and mutator scheduler.
- **Implementation.** We implement our approach as a practical tool based on the Jimple-level mutation via *Soot* [12] with the source code available in our GitHub page [13].
- **Evaluation.** We evaluate *JITfuzz* under multiple experimental setups, and the experimental results suggest that *JITfuzz* outperforms *Classming* by 27.9% and *JavaTailor* by 18.6% in terms of the code coverage. In addition, *JITfuzz* successfully detects 36 previously unknown bugs where 23 are JIT bugs.

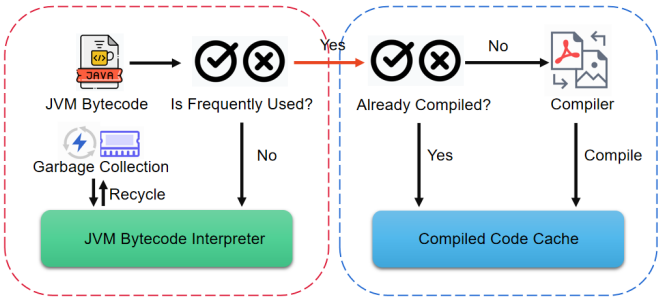

Fig. 1: The workflow of JVM JIT

II. Background

In this section, we give an overview on the features of the basic mechanism of JVM JIT and the challenges of fuzzing JITs to motivate our work.

A. JVM JIT

In JVM, the Just-in-time compiler (JIT) selects code (e.g., frequently used methods) running on JVM and compiles them into the native machine code to accelerate the execution for target programs. Figure 1 illustrates the workflow of JIT. For a given class file, when JVM executes a method, it first verifies whether such a method is frequently used. If so, the method is then compiled into native code by JIT and stored in the compiled code cache for direct execution. Otherwise it is regularly executed on the JVM bytecode interpreter. Since all such selected methods only need to be compiled once, JIT can significantly advance the execution efficiency of target programs. Note that JIT can also be explicitly activated by specifying the JVM command as `java -Xcomp cls` for a given class `cls`.

JIT adopts multiple optimization techniques [8] to optimize program compilation performance at runtime for realizing efficient JVM execution via control flow analysis. Specifically, many widely-used JVMs, e.g., OpenJDK [14], OpenJ9 [15], OracleJDK [16], and JRockit [17], adopt common optimization techniques such as function inlining [18], simplification [19], escape analysis [20], and scalar replacement [21]. More specifically, function inlining refers to merging the small-scale functions into their callers to accelerate the frequent function calls. Simplification refers to using an equivalent but simpler expression to replace the given expression for improving runtime efficiency. Escape analysis refers to identifying the dynamic scope of objects and determine whether to allocate them on the Java heap or replace them with constants, i.e., escape. Note that escape analysis can be implemented in multiple levels, including *GlobalEscape* (i.e., objects escape globally), *ArgEscape* (i.e., objects escape within the same thread) and *NoEscape* (i.e., objects do not escape) [22]. Scalar replacement is one typical solution of the escape analysis in the *ArgEscape* level, i.e., replacing objects in the Java heap within the same thread.

B. Motivation

JIT has been demonstrated to significantly impact the runtime performance of JVM-based applications [23] and thus strongly recommended by industrial developers [24]. While testing JITs to ensure their correctness is essential, there exists no dedicated testing technique for such specific purpose to our best knowledge. Albeit the existing general-purpose JVM fuzzers [25, 7, 9] can potentially expose JIT bugs occasionally, they encounter severe challenges to prevent them from effectively exposing the JIT bugs. In particular, when *ClassFuzz* [25] randomly manipulates (e.g., deletes or inserts) instructions, it can also generate vast invalid/illegal seeds which essentially lead to testing ineffectiveness. Although *Classming* [7] aims at improving over *ClassFuzz* for generating more valid seeds by intentionally breaching the variable dependencies to expose erroneous data flows, it potentially causes early-terminated JVM executions, i.e., insufficient testing of JITs. Meanwhile, noticing that *JavaTailor* [9] demands a preset database containing the Java programs executed to trigger JVM bugs, applying *JavaTailor* to specifically expose JIT bugs can be potentially challenging since the size of the JIT bug datasets are often limited and can only cover a small subset of possible bugs. To address these issues, it is essential for a fuzzer aiming at extending the usage of the JIT optimization techniques, which can be intuitively realized by proposing the fuzzing strategy to both advance the activation of the JIT optimization techniques and mutate control flows while preventing early termination of their associated testing runs.

Many fuzzers, e.g., AFL [2], MOPT [4], and Neuzz [26], adopt code coverage as guidance to facilitate bug/vulnerability exposure, i.e., generating mutants and retaining them as seeds for further mutations if they are executed to increase/optimize code coverage of target programs. However, code coverage can hardly be applied for general-purpose JVM testing techniques because JVMs are likely to cause non-deterministic coverage at runtime due to their adopted mechanisms, e.g., parallel compilation and on-demand garbage collection [7]. On the other hand, we also notice that code coverage can be more deterministically captured for JITs. Therefore, it is plausible and potentially beneficial to adopt code coverage to guide our JIT testing.

III. Approach

We propose *JITfuzz*, a coverage-guided fuzzer for JVM JIT with two mutator types and a mutation scheduler. In particular, *JITfuzz* is implemented with Jimple-level instructions provided by *Soot* [12] which is a framework for analyzing and transforming JVM-based applications. Figure 2 presents the overall workflow of *JITfuzz*. Typically, Initialized with a *seed corpus*, *JITfuzz* iterates each *seed* to generate mutants under a given time budget. For each iteration, *JITfuzz* determines its mutation limit (i.e., the number of mutators applied to the given seed) according to the collected coverage updates. Note that in this paper, *JITfuzz* develops four optimization-activating mutators and two control-flow-enriching mutators to facilitate the usage

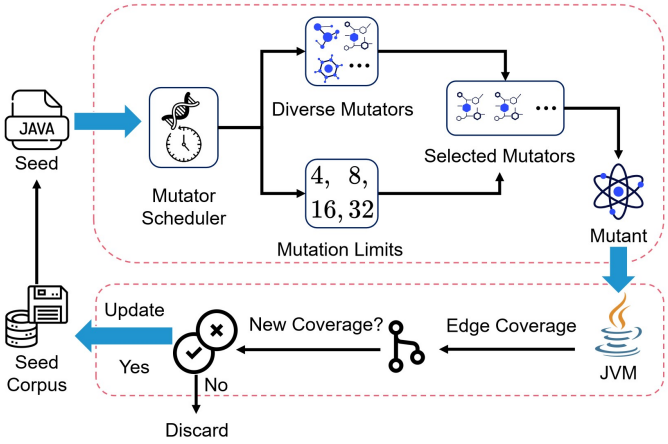

Fig. 2: The framework of *JITfuzz*

of JIT optimization techniques and enrich the control flows. Accordingly, *JITfuzz* adopts a mutator scheduler to select and schedule the mutators by the mutation limit based on a lightweight dynamic optimization algorithm to optimize the runtime code coverage. Eventually, if such resulting mutant increases code coverage in the target JIT, it is added to the *seed corpus* for further mutations.

A. Mutators

As mentioned in Section II, JIT adopts multiple optimization techniques to strengthen the runtime compilation performance of JVM-based programs. Many such optimization techniques involve complicated mechanisms, e.g., code analysis and semantics-preserving code transformations, potentially having the defects which can cause erroneous program executions. Intuitively, extensively applying such optimization techniques can advance JIT bug exposure. Accordingly, we determine to design optimization-activating mutators. Specifically, we first analyze typical JIT optimization techniques, e.g., simplification [19] and escape analysis [22, 27], and then derive the strategies which aim for extensively triggering the corresponding optimization techniques. As a result, such strategies are adopted as the optimization-activating mutators.

We further realize that mutating control flows in seeding class files for executing JIT can potentially advance the effectiveness of JIT fuzzing due to the following reasons. First, control-flow analysis can potentially advance the JIT optimization techniques by identifying where and how to optimize JVM-based programs [8]. Second, control-flow analysis can facilitate the correct compilation from Java programs into native machine code by verifying correctness of semantics [28], etc. At last, existing work [29, 30, 31, 7] also demonstrate that diversifying control flows in running programs can significantly increase fuzzing performance, e.g., coverage. Accordingly in *JITfuzz*, we propose a set of control-flow-enriching mutators including one statement-wrapping mutator to strengthen the usage of basic blocks and one transition-injecting mutator to strengthen the usage of their transitions respectively based on the control flow of a given seed. In addition, they both are designed to preserve semantics

correctness of target programs for preventing early-terminated testing runs, e.g., causing no verification errors.

Note that prior to proposing the mutators, one should adjust the setups of running JVMs such that JITs can be explicitly launched. To this end, we use the JVM command `java -Xcomp cls` for a given class `cls`.

1) Optimization-activating mutators: While optimization-activating mutators can be proposed in accordance with each existing optimization technique, exhaustively designing them can be cost-ineffective. In this paper, we design four optimization-activating mutators corresponding to the representative optimization techniques in the existing JITs. i.e., function inlining [18], simplification [19], escape analysis [20], and scalar replacement [21] which are commonly adopted by the JITs from diverse well-recognized JVMs [14, 32, 33, 34]. Table I shows the details of the optimization-activating mutators via rewrite rules [35].

Function-inlining-activating mutator. Noticing that function inlining refers to merging the instructions of small-scale functions into their callers to reduce the cost of function calls, our function-inlining-activating mutator is proposed to replace a randomly selected instruction with a function where only such instruction is contained, as Rule 1. More specifically, given an expression $\alpha\ op\ \beta$ (op denotes a binary operator), we create a new function $f(x, y)$ which returns the expression $x\ op\ y$. Consequently, we mutate the original expression $\gamma = \alpha\ op\ \beta$ as its corresponding transformation $\gamma = f(\alpha, \beta)$ in the given program context e. For instance, Example 1 shows that the original instruction `return i0 + i1` is mutated by a function `inline` containing it in order to facilitate the application of function inlining and test its capacity of merging small-scale functions into their callers.

Simplification-activating mutator. Noticing that simplification refers to simplifying an arithmetic expression, the simplification-activating mutator replaces a simplistic arithmetic expression as a semantics-preserving yet complicated expression. As in Rule 2, we update the original expression $\alpha\ op\ \beta$ with its semantics-preserving expression $\alpha\ op\ \beta + 0$ and generate an expression $expr$ calculated to be zero. Correspondingly, we mutate the original expression $\gamma = \alpha\ op\ \beta$ as $\gamma = \alpha\ op\ \beta + expr$. To illustrate, Example 2 shows that we mutate the instruction `i2 = i0 + i1` by adding and subtracting a randomly generated integer `i3` at the same time to facilitate the application of simplification on the expression.

Scalar-replacement-activating mutator. Scalar replacement essentially refers to investigating whether a stack variable can replace an object allocated in the heap in order to save memory resources. Accordingly, our scalar-replacement-activating mutator is proposed to replace stack variables with objects in the heap on target programs. Specifically, Rule 3 demonstrates that we first create an object obj and assign an existing variable α as its field, and then replace α with the field $obj.field$ in any original expression $\alpha\ op\ \beta$. Example 3 shows that we create a `Digit` object `r0` to mutate the constant integer `0` in the original instruction to be its associated value stored in `r0.integer` so as to facilitate the application of scalar replacement. Note that such mutator can also be used for the escape analysis in the *ArgEscape* level [22] when JIT verifies whether an object in the heap has side effects or not.

Algorithm 1 Statement-wrapping Mutator

```
   Input : I, seed, Cnt
   Output : mutant
 1: function DefaultControlFlow
 2:     if Cnt ≥ LIMITATION then
 3:         return mutant ← seed
 4:     strategy ← randomly select 0 or 1
 5:     if strategy == 0 then
 6:         expr ← "if (true) {I;}"
 7:     if strategy == 1 then
 8:         expr ← "loop (limit) {I; limit -= 1;}"
 9:     mutant ← update with expr in the seed
10:     Cnt ← Cnt + 1
11:     return DefaultControlFlow(expr, mutant, Cnt)
```

Escape-analysis-activating mutator. To facilitate the escape analysis in the *GlobalEscape* level [22], we also design an escape-analysis-activating mutator that replaces a local object α with the static field of the object $this.field$, which is referred by `this` pointer, as Rule 4. In Example 4, we first create the local object and assign it as the static field of `this` object, and then reassign the reference `i0` to `this.object`. Thus we create a *GlobalEscape* scenario to access the original local object `i0` via the static field `this.object` to facilitate the application of escape analysis.

2) Control-flow-enriching mutator: In this paper, we propose a set of control-flow-enriching mutators to enrich the program control flows for augmenting the fuzzing effectiveness. Note that while the existing JVM fuzzers *Classming* and *ClassFuzz* also adopt control-flow mutators, applying them can easily cause early-terminated testing runs by generating random transitions and fail to expand the size of control flows, e.g., increase the number of basic blocks.

Statement-wrapping mutator. Intuitively, to increase the number of basic blocks in the existing control flows without devastating their executions, one can design *if(true)* statements and/or *loop(limit)* statements to contain the existing program statements. Accordingly, we design a statement-wrapping mutator to wrap a given statement within *if* and/or *loop* statement(s). Algorithm 1 presents the details of applying a statement-wrapping mutator under an input instruction $\mathcal{I}$, its associated *seed*, and a counter *Cnt* denoting the runtime recursion depth. If *Cnt* exceeds threshold *LIMITATION*, the real-time resulting *mutant* is returned (lines 2 to 3). Otherwise, we randomly choose a design strategy to generate a new expression *expr* to contain $\mathcal{I}$ (line 4). One is to generate an *if(true)* block (lines 5 to 6). The other is to generate a *loop(limit)* block (lines 7 to 8). Then the *mutant* is derived by updating the *seed* with the resulting *expr* (line 9) followed by updating *Cnt* (line 10). Note that the above operations are recursively executed (line 11).

Transition-injecting mutator. We propose a transition-injecting mutator to enrich the transitions among basic blocks. While we simply select two random basic blocks for generating their transition, we also realize that without ensuring the

TABLE I: Optimization-Activating mutators

Optimization techniques	Rules for Jimple-level optimization-activating mutators	Example

Function Inlining	**Rule 1:**
	$[\![e[\gamma = \alpha\ op\ \beta]\ end]\!] \rightarrow$
	$[\![let\ function\ f(x,y) = x\ op\ y\ in\ e[\gamma = f(\alpha,\beta)]\ end]\!]$
		Example 1:
		`-int i2 = i0 + i1;`
		`+public int inline(int i0, int i1) {`
		`+  return i0 + i1;`
		`+}`
		`+int i2 = inline(i0, i1);`

Simplification	**Rule 2:**
	$[\![e[\gamma = \alpha\ op\ \beta]\ end]\!] \rightarrow$
	$[\![let\ expr = 0\ in\ e[\gamma = \alpha\ op\ \beta + expr]\ end]\!]$
		Example 2:
		`-int i2 = i0 + i1;`
		`+int i3 = new Random().nextInt();`
		`+int i2 = (i0 + i3) + (i1 - i3);`

Scalar Replacement & Escape Analysis (ArgEscape level)	**Rule 3:**
	$[\![e[\gamma = \alpha\ op\ \beta]\ end]\!] \rightarrow$
	$[\![let\ obj.field = \alpha\ in\ e[\gamma = obj.field\ op\ \beta]\ end]\!]$
		Example 3:
		`-int i0 = 0;`
		`+Digit r0 = new Digit(0);`
		`+int i0 = r0.integer;`

Escape Analysis (GlobalEscape level)	**Rule 4:**
	$[\![e[\gamma = \alpha\ op\ \beta]\ end]\!] \rightarrow$
	$[\![let\ this.field = \alpha\ in\ e[\gamma = this.field\ op\ \beta]\ end]\!]$
		Example 4:
		`-Object i0 = new Object();`
		`+this.object = new Object();`
		`+Object i0 = this.object;`

correct execution of the associated target programs, it is likely to cause early-terminated program execution, i.e., the verification error caused by undefined variables. Figure 3 presents a real-world illustrative example (Since the code is complex, we demonstrate the complete code and its corresponding labels in [36] due to page limit) where the dark solid lines denote the existing transitions between basic blocks. Assume by applying the statement-wrapping mutators, a transition is established from `f8` to `f5` (denoted as the red dash line) and thus leads to an execution path `[f1, f7, f8, f5]`. However, since `f5` depends on variables `sampleIndex` and `samplePos` [36] defined in `f3` instead of any of its predecessors on execution path `[f1, f7, f8, f5]`, this transition definitely causes an undefined variable error and prevents further testing on the deep states of JIT, e.g., the optimization techniques. Therefore, when designing the transition-injecting mutator, we also need to resolve the potential dependency issues so as to facilitate testing deep states of target programs. Note that according to *Soot* [12], any variables used by basic block β is defined either in β or its dominators. Therefore, to prevent undefined variable errors when creating a transition between two randomly chosen basic blocks (represented as $\alpha \rightarrow \beta$), it is essential to redirect the transition from α to a dominator of β that define all variables possibly used in β. However, there can be a possible side effect that if massive such transitions are redirected to the same dominator such as the identical entry basic block among similar seeds, their remaining partial control flows can be quite alike or even identical, causing limited program execution spaces and thus hindering program state exploration. Therefore in this paper, for a randomly generated basic block transition $\alpha \rightarrow \beta$, we determine to redirect it from α to the closest dominator of β which causes no undefined variable errors when applying the transition-injecting mutator.

Algorithm 2 illustrates our transition redirection mechanism when applying the transition-injecting mutators. Given an input seed, we first construct its control-flow graph and identify its entry basic block (lines 2 to 3), and then generate a *directed*

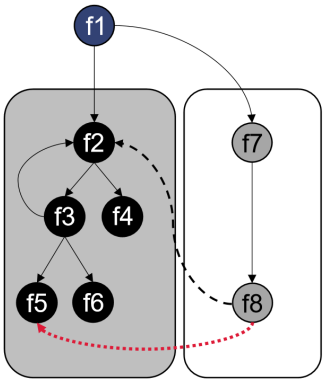

Fig. 3: An example [36] for illustrating the transition-injecting mutator

CFG by deleting all the transitions created by executing *loop* expressions or applying transition-injecting mutators (line 4) on top of the original CFG. After randomly selecting the source basic block `src` and the sink basic block `sink` (lines 5 to 6), we store all the dominators of `src` in order as the list `srcPre` (lines 9 to 11). Next, we also store all the dominators of `sink` in order as the list `sinkPre`. Accordingly, we identify the closest common dominator from `sinkPre` and `srcPre` as `sinkFiC` (lines 12 to 17). Finally, we identify the immediate post-dominator of `sinkFiC` in the set `sinkPre` as `sinkNew` and create a transition from `src` to `sinkNew` to generate a mutant (lines 18 to 19). For example in Figure 3, we first assume `f8` is `src` and `f5` is `sink`. Next, we obtain the closest common dominator for `f8` and `f5`, i.e., `f1`. Then, `f2` is identified as the immediate post-dominator of `f1` among all the dominators of `f5`. At last, we create transition `f8`→`f2` to generate a mutant.

We then discuss why Algorithm 2 can prevent early termination of program executions and limited program state exploration, i.e., the sink dominator `sinkNew` selected by Algorithm 2 is the closest dominator of `sink` causing no undefined variable error. First, assume that redirecting the transition from `src` to `sinkNew` incurs an undefined variable

Algorithm 2 Transition Redirection Mechanism

Input: *seed*
Output: *mutant*

1: **function** REDIRECTBASICBLOCKTRANSITION
2: *controlFlowGraph* ← *obtainCFG(seed)*
3: *entry* ← *identifyEntry(seed)*
4: *directedCFG* ← *deleteSelectedEdges(controlFlowGraph)*
5: *src* ← *randomlySelectBasicBlock(directedCFG)*
6: *sink* ← *randomlySelectBasicBlock(directedCFG)*
7: *srcPre, sinkPre, sinkFiC* ← {}, {}, {}
8: *srcNxt, sinkNxt* ← *src*, *sink*
9: **while** *entry* ∉ *srcPre* **do**
10: *srcNxt* ← *getDominator(srcNxt, directedCFG)*
11: *srcPre* ← *srcPre* ∪ {*srcNxt*}
12: **while** *entry* ∉ *sinkPre* **do**
13: *sinkFiC* ← *sinkPre* ∩ *srcPre*
14: **if** *sinkFiC* ≠ ∅ **then**
15: **break** ▷ sinkFiC only contains one dominator
16: *sinkNxt* ← *getDominator(sinkNxt, directedCFG)*
17: *sinkPre* ← *sinkPre* ∪ {*sinkNxt*}
18: *sinkNew* ← *get the immediate post-dominator of sinkFiC from sinkPre*
19: *mutant* ← *create a transition from src to sinkNew*
20: **return** *mutant*

error in the original error-free program execution, i.e., a variable is used without a definition in `sinkNew` after redirection. Then we infer that this variable can be accessed by `sinkFiC` since `sinkFiC` dominates `sinkNew` in `sinkPre`. Accordingly, this undefined variable error can be spread in `sinkFiC` (otherwise, `sinkFiC` is obliged to define variables to prevent the undefined variable error in `sinkNew`), contradicting our assumption. Thus, redirecting the transition to `sinkNew` is ensured to incur no undefined variable error. Next, we discuss why `sinkNew` is the closest dominator of `sink` which the transition can be redirected to without causing any undefined variable error. Similarly, assume there is a post-dominator of `sinkNew` which the transition can be redirected to without causing any undefined variable error. Since redirecting the transition from `src` to the post-dominator of `sinkNew` does not cause any undefined variable error, we infer that `sinkNew` is not allowed to define new variables (otherwise, it is possible that the variable is only defined in `sinkNew` which is not included in the execution after the transition redirection, causing an undefined variable error in its post-dominator). Such inference contradicts the fact that `sinkNew` is allowed to define new variables as a dominator of `sink`. Thus, `sinkNew` is ensured to be the closest dominator of `sink` causing no undefined variable error.

B. Mutator Scheduler

After proposing multiple mutator types to strengthen the usage of the optimization techniques in JITs in *JITfuzz*, how to aggregate their strengths to optimize their overall effectiveness becomes our next challenge. To this end, intuitively, an optimization guide is essential. Note that while JVMs are likely to cause non-deterministic coverage at runtime due to their adopted mechanisms, e.g., parallel compilation and on-demand garbage collection [7] (such that coverage usually cannot be applied as a guide for testing JVMs), coverage updates can be more deterministically captured for JITs. Therefore, we determine to adopt the runtime coverage updates of target programs for guiding our mutator scheduling plans.

In this paper, we build our mutator scheduler upon the UCB-1 algorithm [37], a lightweight algorithm which constructs an optimistic guess to the expected payoff of each action and picks the action with the optimal payoff to guide future iterative executions. The UCB-1 algorithm is adopted to schedule the mutators in *JITfuzz* due to the following reasons. First, in *JITfuzz*, scheduling mutators to optimize their aggregated effectiveness at runtime essentially is a stochastic optimization problem which exploits limited knowledge (i.e., runtime coverage). Notably, the UCB-1 algorithm is proposed to exactly address such stochastic optimization problem and has been widely adopted for similar tasks [37, 38, 39]. Next, scheduling mutators for fuzzing essentially demands limited overhead such that adequate computing resources can be leveraged for key technique components, e.g., mutations, program executions, and coverage collections. Notably, the UCB-1 algorithm yields rather limited overhead, i.e., quickly adjusting the mutator options based on runtime coverage updates, to approach the optimal solutions for each iterative execution. As a result, *JITfuzz* utilizes the UCB-1 algorithm to schedule the mutators according to runtime coverage updates. In particular, for a given seed, *JITfuzz* first identifies a mutation limit to determine how many mutators should be scheduled. Similar to AFL [2], *JITfuzz* adopts multiple mutation limit options (four in our paper, i.e., 4, 8, 16, 32). Next, by the scheduled mutation limit, *JITfuzz* repeatedly selects one mutator out of all the six possible options (i.e., four optimization-activating mutators and two control-flow-enriching mutators). More specifically, the mutators can be scheduled via Equation 1 where $result(t)$ denotes both the mutation limit result and the corresponding mutator result at the t-th iteration.

$$result(t) = \arg\max_{j} \left(\frac{1}{t_j}\sum_{i=1}^{t_j} x_{ji} + \sqrt{\frac{2\ln(t-1)}{t_j}}\right) \quad (1)$$

In Equation 1, t_j denotes the total number that the j-th mutator/limit option has been selected till the t-th iteration, and x_{ji} refers to the reward of the j-th mutator/limit option in the i-th iteration. Accordingly, the selected mutators are applied to the given seed in turn for generating the mutants at the t-th iteration. Meanwhile, the obtained coverage is recorded to update the scheduler as a reward, which is one if the coverage is increased and zero otherwise.

C. Discussion

We consider the concept of *JITfuzz* can be inspiring for more fuzzing domains where multiple components are included in one testing target. In particular, when proposing mutants to cope with the target program as a whole instead of their individual components, it is possible that the generated mutators fail to fully access their components and thus compromise the fuzzing effectiveness. On the other hand, applying the mutators for addressing individual components only might still

incur testing ineffectiveness, since they are highly likely to cause divergent testing effects respectively [4]. Therefore, it is promising to include an additional mutator scheduler to augment their collective power.

IV. Evaluation

In this section, we conduct a set of experiments to evaluate the effectiveness of *JITfuzz* on a real-world benchmark suite composed of 16 projects. In particular, we first collect 6 projects commonly adopted from prior work [7, 9] and 10 additional popular open-source Java projects in GitHub [10]. Next, we compare *JITfuzz* with state-of-the-art mutation-based JVM fuzzer *Classming* [7] and generation-based JVM fuzzer *JavaTailor* [9] in terms of the edge coverage results obtained from JIT of our target JVM, and evaluate the effectiveness of different components of *JITfuzz*. In particular, we attempt to answer the following research questions:

- ***RQ1:*** *Is JITfuzz effective in fuzzing JIT?*
- ***RQ2:*** *Are the different components of JITfuzz effective in terms of ablation study*?

Moreover, we report and analyze the bugs on our adopted benchmark exposed by *JITfuzz*. Note that all the evaluation details are presented in our GitHub page [13].

A. Benchmark Construction

In this paper, we define a set of rules to collect influential real-world Java projects in GitHub to form our benchmark for our evaluation. In particular, we search with the keyword "Java" on GitHub, and then randomly select 10 projects with decent star number (larger than 100) and LoC numbers (larger than 10k). In addition, we select all 6 projects from state-of-the-art *JavaTailor* which are all excerpted from the dacapo benchmark [40] with 102 stars. Table II demonstrates the detailed information of our benchmark suite where the top 6 projects are adopted by *JavaTailor*. Furthermore, for each of these projects, we randomly select one of the ten classes with the highest cyclomatic complexity [41] as the seed program where the cyclomatic complexity refers to the number of linearly independent paths through the source code of a class [42, 43, 44, 45].

TABLE II: Benchmark information

Project	Stars	LoC	Initial class(seed)
avrora	102	111k	...avrora/Main.class
eclipse		26.4k	...EclipseStarter.class
pmd		197.4k	...pmd/PMD.class
jython		360.7k	...python/util/jython.class
fop		331.3k	...fop/cli/Main.class
sunflow		28.5k	...sunflow/Benchmark.class
hutool [46]	23.5k	265.7k	..core.text.PasswdStrength.class
javapoet [47]	9.7k	12.4k	...ClassName.class
mybatis-3 [48]	17.5k	161.3k	...ibatis.parsing.GenericTokenParser.class
zxing [49]	29.9k	219.3k	...zxing.qrcode.encoder.Encoder.class
fastjson [50]	24.8k	103.9k	...fastjson.JSON.class
guice [51]	11.3k	110.6k	...inject.spi.InjectionPoint.class
commons-text [52]	242	54.7k	...commons.text.numbers.ParsedDecimal.class
rocketmq [53]	17.8k	178.2k	...rocketmq.filter.util.BloomFilter.class
spark [54]	9.3k	23.1k	spark.resource.UriPath.class
vert.x [55]	3.3k	215.4k	...vertx.core.json.JsonArray.class

B. Environment Setup and Implementation

We perform our evaluations on a work station, with AMD EPYC 7H12 CPU and 256 GB memory. The operating system is 64-bit Ubuntu 18.04.5 LTS. We choose HotSpot (Java 19) [14] as our target JVM to obtain the coverage results and set the *LIMITATION* for Algorithm 1 to two. Note that the results of more *LIMITATION* setups are presented in our GitHub link [13] due to page limit. Moreover, following prior work [2, 26, 56], we adopt the edge coverage [57] obtained from JIT to reflect the effectiveness of our studied techniques. To collect runtime edge coverage, we first obtain all the source files of JIT only. Next, we utilize the partial instrumentation tool from AFL++ [58] to instrument such source files for runtime edge collection. Note that JVM does have non-deterministic coverage such as garbage-collection mechanism [7]. However, since we only include the source files of JIT, such issues are mitigated while we collect the edge coverage information during fuzzing JIT.

All our experiments are run for 24 hours following prior work [56, 26, 59]. Note that all the experimental results are averaged from five runs to reduce the impact of randomness.

C. Result Analysis

1) RQ1: the effectiveness of JITfuzz: Table III demonstrates the evaluation results of *JITfuzz*, *JavaTailor* and *Classming* in terms of edge coverage.

Overall, we observe that *JITfuzz* can significantly outperform *Classming* in terms of the edge coverage. Specifically, *JITfuzz* can explore averagely 34,939 edges, while *Classming* only explores 27,306 edges, i.e., *JITfuzz* explores over 27.9% more edges than *Classming*. Meanwhile, we also find that *JITfuzz* outperforms *JavaTailor* by 18.6% (34,939 edges vs. 29,451 edges). Moreover, *JITfuzz* can significantly outperform *Classming* and *JavaTailor* on each individual project (from 4.9% to 1.8× for *Classming* and from 3.0% to 98.4% for *JavaTailor*). Such results altogether reflect that *JITfuzz* achieves significant edge coverage advantages over *Classming* and *JavaTailor*.

We also apply the Mann-Whitney U test [60] to illustrate the significance of *JITfuzz* in Table III. We can observe that the p-values of *JITfuzz* comparing with other studied fuzzers in terms of average edge coverage are far below 0.05, which indicates *JITfuzz* outperforms all studied fuzzers significantly.

> *Finding 1: JITfuzz is more effective than Classming and JavaTailor by exploring 27.9% and 18.6% more edges on average.*

2) RQ2: Effectiveness of each component: In this section, we conduct a set of ablation studies to evaluate the effectiveness of the key technical components in *JITfuzz*.

Effectiveness of the mutators. to perform ablation studies on the effectiveness of each mutator, we build the following six variant techniques of the original *JITfuzz* by disabling the corresponding mutators, i.e., *JITfuzz*$_{-inline}$, *JITfuzz*$_{-simp}$, *JITfuzz*$_{-scalar}$, and *JITfuzz*$_{-escape}$, *JITfuzz*$_{-wrap}$, and

JITfuzz$_{-trans}$ which respectively disables the function-inlining-activating mutator, the simplification-activating mutator, the scalar-replacement-activating mutator, the escape-analysis-activating mutator, the statement-wrapping mutator, and the transition-injecting mutator. Accordingly, we can derive the effectiveness of a mutator by comparing the performance of its associated technique variant with the original *JITfuzz*. Table III demonstrates the edge coverage results of *JITfuzz* and our studied technique variants. In general, we can observe that *JITfuzz* outperforms each variant averagely from 12.2% to 26.9% in terms of edge coverage, which is rather substantial. Moreover, we also find that all the variant can outperform *Classming* in terms of edge coverage from 0.8% to 14.0% respectively and three variants outperform *JavaTailor* from 2.6% to 5.7%, which delivers the fact that the framework of *JITfuzz* is robust even with partial mutators. Such results suggest that all mutators are effective and integrating them together optimizes the performance in exploring edges for JIT.

Finding 2: Each mutator of JITfuzz is effective and integrating them optimizes the performance of exploring edges.

Effectiveness of the transition redirection mechanism. We further investigate the effectiveness of the transition redirection mechanism for applying the transition-injecting mutator. As in Algorithm 2, the transition-injecting mutator redirects a randomly generated transition from the source to a dominator of the sink to prevent using undefined variables. Accordingly, we build a technique variant *JITfuzz*$_{randtr}$, which simply creates transition directly for two randomly chosen basic blocks without applying any redirection while retaining all other proposed mutators in this paper.

We can observe from Table III *JITfuzz* significantly outperforms *JITfuzz*$_{randtr}$ by 28.6% on average in terms of edge coverage, indicating that redirecting transitions is essential to facilitate the efficacy of the transition-injecting mutator.

Finding 3: Transition redirection mechanism is essential for the transition-injecting mutator in augmenting its edge coverage performance.

Effectiveness of the mutator scheduler. To investigate the effectiveness of mutator scheduler, we build a technique variant *JITfuzz*$_{randsch}$, which randomly schedules the mutators and the mutation limit in each iterative execution instead of applying the coverage-guided mutator scheduler.

In general, we can observe from Table III that mutator scheduler is effective since *JITfuzz* outperforms *JITfuzz*$_{randsch}$ significantly by 10.2% (34,939 vs. 31,700 explored edges) on average in terms of edge coverage. More specifically, we can further observe that *JITfuzz* outperforms *JITfuzz*$_{randsch}$ consistently upon all the benchmark projects. Such results indicate that our adopted mutator scheduler is rather powerful in strengthening the effectiveness of JIT fuzzing.

Finding 4: Adopting mutator scheduler can significantly improve edge exploration for JITfuzz by scheduling mutators and its mutation limit.

D. Bug Report and Analysis

JITfuzz is effective in exposing multiple real-world JIT/JVM bugs where a bug is defined as a defect within a specific JVM version in this paper. In our paper, a bug is exposed if a seed 1) triggers any JVM crash, or 2) incurs different outputs among different JVMs (e.g., one JVM normally terminates while another reports an exception). After careful manual analysis, we then report the potential bugs as well as the corresponding seeds to the developers. Specifically, we apply the seeds generated by *JITfuzz* in our evaluation to run multiple JVMs, i.e., different versions of OpenJ9 [15], OpenJDK [14], and OracleJDK [16], for exposing their bugs. Note that we tend to include their recent versions since they are typically adopted by a large-scale collection of real-world projects, i.e., potentially having more impact than the older ones.

Table IV demonstrates the detailed information where we have successfully detected 36 JVM bugs. After reporting them to the corresponding JVM developers, 27 of them have been confirmed and 16 have been fixed. More specifically, 23 of them are JIT bugs, 18 have been confirmed, and 7 are fixed by the developers. Note that none of the bugs can be detected by *Classming* or *JavaTailor*. To illustrate, we observe that executing many seeds adopted by *Classming* fail to activate JIT optimization techniques with the abrupt termination. Meanwhile, the database adopted by *JavaTailor* which contains the Java programs exposing JVM bugs fails to contain sufficient JIT bugs for reference. We then introduce four typical bugs exposed by *JITfuzz* as follows.

1) JIT segmentation fault: We reported a vulnerability [61] on the C2 compiler [62]—a specific JIT compiler in HotSpot (OpenJDK), which affects several OpenJDK versions, including 7u351, 8, 11, 17.0.2, 18, and 19. It was assigned with a bug ID JDK-8283441 and has been fixed later. This vulnerability is exposed by running the original JUnit tests with a seed generated from `JSON.class`. In particular, the affected JVMs crashed due to a segment fault within `ciMethodBlocks::make_block_at(int)`, as in Figure 4.

The developers located the issue to two methods in the seed whose bytecodes end abruptly with unreachable basic blocks, as shown in Figure 5. They found that HotSpot builds control flow graphs for unreachable basic blocks where JIT fails to validate. As a result, by compiling unreachable basic blocks, JIT accesses invalid memory, causing the segmentation fault. Eventually, they fixed this issue as follows:

"The new verifier checks by bytecodes falling off the end of the method, and the old verify does the same, but only for reachable code. So we need to be careful of falling off the end when compiling unreachable code verified by the old verifier."

TABLE III: Effectiveness of the *JITfuzz* mutators

Benchmark	*JITfuzz*	*Classming*	*JavaTailor*	Variants by Disabling Mutators						Other Variants	
				$JITfuzz_{-scalar}$	$JITfuzz_{-escape}$	$JITfuzz_{-simp}$	$JITfuzz_{-inline}$	$JITfuzz_{-wrap}$	$JITfuzz_{-trans}$	$JITfuzz_{randtr}$	$JITfuzz_{randsch}$
hutool	37,006	32,994	33,345	34,209	34,483	35,276	32,018	29,378	30,374	31,367	35,038
javapoet	36,178	33,948	35,138	33,675	30,990	34,296	34,135	32,381	32,070	32,465	33,599
mybatis	29,209	18,708	21,642	23,165	22,354	25,358	24,126	22,874	24,642	16,560	26,064
zxing	36,184	28,350	33,406	35,214	30,161	34,931	35,680	34,474	28,731	29,682	32,156
fastjson	36,222	26,121	26,956	30,114	34,367	34,963	33,715	33,269	28,711	26,341	32,874
guice	36,852	30,757	32,257	29,805	31,241	30,571	24,521	26,937	28,227	26,734	32,144
commons-text	36,863	35,147	34,058	35,075	36,075	36,881	34,503	32,958	34,039	33,129	33,039
rocketmq	32,808	29,043	30,910	26,977	28,457	24,097	26,241	26,012	24,060	24,782	30,422
spark	33,936	12,130	17,109	28,145	24,270	24,854	32,568	28,613	26,279	25,638	29,695
vert.x	34,056	24,162	25,598	28,056	23,806	31,199	22,785	27,322	25,302	26,986	32,238
avrora	36,835	29,339	32,634	28,879	28,208	36,275	32,539	28,344	24,598	28,150	33,073
eclipse	33,057	30,299	29,692	29,518	26,841	31,556	28,035	24,086	24,933	24,925	28,692
pmd	36,985	28,898	32,505	30,263	30,123	29,788	30,522	27,963	28,801	25,441	32,623
jython	35,284	25,827	29,060	29,443	31,155	30,630	29,892	32,110	25,226	28,139	33,102
fop	32,713	22,972	26,782	27,439	27,934	29,878	31,426	29,269	28,880	26,912	30,453
sunflow	34,846	28,202	30,133	33,632	26,595	27,584	30,553	30,851	25,713	27,317	32,002
average	**34,939**	**27,306**	**29,451**	**30,225**	**29,191**	**31,133**	**30,203**	**29,177**	**27,536**	**27,160**	**31,700**
p-value	**N/A**	**0.004**	**0.004**	**0.004**	**0.004**	**0.004**	**0.004**	**0.004**	**0.004**	**0.004**	**0.004**

TABLE IV: Issues found by *JITfuzz*

JVMs	# Issues Reported		# Issues Confirmed		# Issues Fixed	
	JIT	Non-JIT	JIT	Non-JIT	JIT	Non-JIT
OracleJDK	1	0	1	0	0	0
OpenJDK	18	5	13	5	7	5
OpenJ9	4	8	4	4	0	4
TOTAL	23	13	18	9	7	9

```
ciBlock *ciMethodBlocks::make_block_at(int bci) {
  ciBlock *cb = block_containing(bci);
  if (cb == NULL ) {
    ciBlock *nb = new(_arena) ciBlock(_method,
   _num_blocks++, bci);
    _blocks->append(nb);
    // segmentation fault
     _bci_to_block[bci] = nb;
    return nb;
  } else if (cb->start_bci() == bci) {
    return cb;
  } else {
    return split_block_at(bci);
  }
}
```

Fig. 4: One C2 segmentation fault bug in HotSpot

2) Dead loop assertion failure: *JITfuzz* discovered a HotSpot vulnerability [61] on JIT caused by an assertion failure, indicating that a dead loop was detected as in Figure 6. The OpenJDK versions 8, 11, 17, 18, 19 and 20 are affected by this vulnerability which has been reported to the developers and assigned with a bug ID JDK-8280126.

The developers confirmed this bug and tried to analyze the corresponding buggy class file but failed by applying the tools provided by OpenJDK. They implemented multiple helper functions to analyze the control structure and inferred that HotSpot may miscalculate the control flow and consider certain nodes to be unreachable. As a result, such nodes

```
416: return
// Unreachable block
417: iinc       5, 1
420: iload      5
422: iconst_2
423: if_icmple  339
(end of bytecode)
```

(a) JSON.config() code snippet

```
623: areturn
// Unreachable block
624: iinc       20, 1
627: iload      20
629: iconst_2
630: if_icmple  336
(end of bytecode)
```

(b) JSON.parseObject() code snippet

Fig. 5: Unreachable basic blocks in the generated class

```
void PhaseGVN::dead_loop_check( Node *n ) {
  if (n != NULL && !n->is_dead_loop_safe() && !n->
   is_CFG()) {
    bool no_dead_loop = true;
    ...
    if (!no_dead_loop) n->dump(3);
    // assertion failure
    assert(no_dead_loop, "dead loop detected");
  }
}
```

Fig. 6: One dead loop assertion in HotSpot

take unexpected data as input and cause a dead loop, e.g., a data node in control flow graph references itself directly or indirectly. Eventually, they decided to defer this issue to JDK 20 due to its complexity with the following feedback:

> *"The difficulty with this bug is that we have many paths that get eliminated, finding the real source of the issue feels like searching for a needle in a haystack."*

3) JIT crash during optimization: We reported an OpenJ9 vulnerability [63] on JIT optimizer, which has been confirmed by the developers. When applying option `optlevel` at the `hot` level or higher, JIT failed to compile a seed generated from `ParsedDecimal.class` due to a segmentation error within `TR_OrderBlocks::peepHoleBranchBlock`, as shown in Figure 7.

```
void TR_OrderBlocks::peepHoleBranchBlock(TR::CFG *
   cfg, TR::Block *block, char *title)
{
...
// crash
TR::Block *fallThroughBlock = fallThroughEntry->
getNode()->getBlock();
```

Fig. 7: Assertion failure during verification

The developers found that a node created by `generalLoopUnroller` is `NULL`. Furthermore, this issue can be bypassed by setting the JVM option `disableGLU` to disable the `general loop unroller`. Eventually, devel-

opers concluded that JIT miscalculated the control flow graph to incorrectly move certain nodes.

> *"However If I add tracing on this method, the crash goes away. I'll instrument the code and see if I can catch the issue in an early stage of the optimization."*

4) Other runtime vulnerabilities: In addition to JIT vulnerabilities, *JITfuzz* also reveals runtime defects. For example, we reported an OpenJ9 vulnerability [64] on the verification stage, which causes OpenJ9 to crash due to an assertion failure.

To locate this issue, the developers reproduced the crash in a debug build and found that the crash occurred when releasing the stackmap frame memory at `/runtime/verbose/errormessagehelper.c`, as shown in Figure 8.

```
releaseVerificationTypeBuffer(StackMapFrame*
    stackMapFrame, MethodContextInfo* methodInfo)
{
   if (NULL != stackMapFrame->entries) {
     PORT_ACCESS_FROM_PORT(methodInfo->portLib);
     // crash
     j9mem_free_memory(stackMapFrame->entries);
  }
}
```

Fig. 8: Assertion failure during verification

The developers further found that OpenJ9 incorrectly built stackmaps for the class file due to its huge size. In particular, OpenJ9 cannot allocate the memory for its locals and stack and the crash occurs when it tries to release the memory of the stackmap frame. The developers replied as follows.

> *"There is no element of 'locals' and 'stack' in the current stackmap frame in which case the code above didn't handle at this point."*

Eventually, the developers fixed this issue by adding additional checks to handle the case with empty locals and stack.

To summarize, by adopting the optimization-activating mutators, the statement-wrapping mutators and the mutator scheduler with easy-to-capture coverage information, *JITfuzz* can expose multiple types of JIT bugs which can be hardly explored by the existing approaches.

V. THREATS TO VALIDITY

The threats to external validity mainly lie in the subjects used in our benchmark. To reduce the threats, we determine to collect important and influential real-world Java projects to form our benchmark. Specifically, we first adopt all the 6 projects from *JavaTailor*. Next, we randomly select 10 GitHub projects with decent star/LoC numbers. We then randomly select one class with high cyclomatic complexity from each project as the seed.

One threat to internal validity lies in the potential flaws in our implementation of *JITfuzz*. To reduce the threat, the first three authors in the paper have been carefully working on the implementation for over one year. We manually reviewed all our implemented code and tested them sufficiently for verifying our implementation. Another threat to internal validity lies in the scalability of optimization-activating mutators for diverse JVM implementations. To reduce the threat, we recognize four commonly-adopted optimization techniques after rigorous studies on multiple existing JVMs and design their corresponding optimization-activating mutators. Moreover, new mechanism-based mutators can be easily proposed for *JITfuzz* when including additional JIT optimization techniques.

The threats to construct validity mainly lie in the metrics used. To reduce the threats, we use a widely-used metric, i.e., the edge coverage, to evaluate our approach. We also deliver detailed bug report from real-world benchmarks to strengthen the evaluation on the real-world applicability of *JITfuzz*.

VI. RELATED WORK

A. Fuzzing

Fuzzing [65] refers to an automated software testing technique that inputs unexpected or random data to programs such that the program exceptions including crashes, failing code assertions, or memory leaks can be exposed and monitored. As a widely-adopted baseline fuzzer, AFL [2] provides a fundamental implementation for coverage-guided fuzzing framework with components such as instrumentation, edge coverage collector, etc. Many existing fuzzers are proposed to enhance the performance of AFL. For instance, Böhme et al. [3] proposed AFLFast to enhance AFL by scheduling the seeds via Markov chain. To explore the rare branches, Lemieux et al. [5] attempted to first identify branches exercised by limited seeds produced by AFL and then invest adequate computation resources to such branches to thoroughly explore target programs. Lyu et al. [4] proposed MOPT to improve the performance of AFL by scheduling existing mutators via the Particle Swarm Optimization (PSO) algorithm. Wu et al. [66] found that combining deep learning and program smoothing can be helpful for fuzzing while it also can be improved by injecting a mechanism for identifying edge properties. They further discovered that the widely-adopted Havoc mechanism potentially dominates the effectiveness of fuzzing strategies [67]. Jiang et al. [68] found that the coordination mode of fuzzing and concolic execution is the dominating factor for the effectiveness of hybrid fuzzers. Furthermore, fuzzing has been widely adopted in multiple domain-specific scenarios. For example, Noller et al. [69] proposed QFuzz to quantitatively evaluate the strength of side channels with a focus on min entropy, which is a measure based on the number of distinguishable observations (partitions) to assess the resulting threat from an attacker who tries to compromise secrets in one try. Andronidis et al. [70] leveraged the power of snapshot of net applications to facilitate the fuzzing efficacy of networking applications. Wu et al. [71] proposed Simulee to fuzz CUDA programs for exposing their synchronization bugs and further fixed them accordingly [72]. Ma et al. [73] proposed PrIntFuzz to fuzz Linux drivers by a contructed device simulator. Zheng et al. [74] proposed EQUAFL to fuzz Linux-based IoT devices via switching emulation types. By

using raw html files as inputs, Song et al. [75] proposed R2Z2 which utilizes differential testing framework to fuzz web browsers. With the help of the debugging feature, Li et al. [76] proposed μAFL to bridge the fuzzing environment on PC and target firmware on microcontroller devices. Recently, a large group of researchers have been focusing on applying fuzzing/testing to the emerging AI/ML systems [77, 78, 79, 80, 81, 82, 83, 84].

Although many existing fuzzers have been proven effective in fuzzing real-world programs, there exists no fuzzer specifically for JVM JITs. In this paper, we propose *JITfuzz*, the first coverage-guided JVM JIT fuzzer which includes multiple effective mutators and a mutator scheduler to expand code coverage and expose real-world JIT/JVM bugs.

B. Compiler and JVM testing

Researchers have spent large effort on compiler testing. To date, a number of techniques have been proposed to automatically generate programs for compiler testing. Yang et al. [30] proposed Csmith, which generates the seeding C programs to explore C compiler while preventing the undefined and unspecified behaviors that might cause testing insufficiency. Reddy et al. [85] proposed RLCheck, which utilizes reinforcement learning to generate diverse valid inputs to explore the programs requiring strict validation on the inputs, e.g., JavaScript engine. Eberlein et al. [86] developed EVOGFUZZ, an evolutionary grammar-based fuzzing approach to optimize the probabilities to generate test inputs that are likely to trigger unexpected behaviors for applications with common input formats (JSON, JavaScript, or CSS3). For JVM testing, Yoshikawa et al. [87] proposed a random Java program generator based on the predefined syntax to expose JVM vulnerabilities by differential testing. Sirer et al. [88] proposed lava to generate Java programs for JVM testing via randomly iterating over the Java grammar productions. Boujarwah et al. [89] utilized the predefined grammar of Java programs to generate semantics-correct test cases to fuzz JVM. More recently, Zhao et al. [9] proposed JavaTailor to generate testing programs by learning information from historical bug-revealing test programs to expose JVM defects.

Compared with the above-mentioned generation-based testing approaches which either generate seeding programs from scratch (e.g., RLcheck [85]), or require program samples with additional efforts (e.g., JavaTailor [9]), the mutation-based testing approaches are usually launched with specifically designed mutators and collected seed corpus. To systematically parse existing real-world code for producing discrepancy-induced programs, Le et al. [90] introduced equivalence modulo inputs (EMI) to mutate seed programs and validate the quality of compilers. Zhang et al. [91] introduced the concepts of the skeletal program enumeration and replaced the variables in the original skeletal program to generate control flows for compiler testing. Sun et al. [92] tested compilers via mutating the live code of the input programs regardless the restriction of only mutating the unreachable regions of input programs. Donaldson et al. [31] developed GLFuzz for testing OpenGL shading language compilers based on semantics-preserving program transformations. Park et al. [93] proposed Die to fuzz the JavaScript engine via the aspect preservation mutators. In terms of testing JVM, Chen et al. [25] proposed classfuzz, which utilizes Markov Chain Monte Carlo [94, 95] to guide mutation via designed mutators. They also proposed classming [7] to leverage the power of manipulating the control flows of seeding class files to test the execution engine of JVM.

Compared with traditional compiler and JVM testing approaches, our proposed *JITfuzz* can be readily applied to test JITs thoroughly with the well-designed mutators and mutator scheduler under given collection of seed inputs.

VII. CONCLUSION

In this paper, we develop a coverage-guided fuzzing framework for JVM JITs, namely *JITfuzz*, which includes four optimization-activating mutators and two control-flow-enriching mutators. Moreover, *JITfuzz* adopts a lightweight mutator scheduler to schedule the mutation limit and the associated mutators for maximizing the overall effectiveness of fuzzing. To evaluate the effectiveness of *JITfuzz*, we construct a benchmark suite with 16 real-world JVM-based projects. Our evaluation results suggest that *JITfuzz* can outperform state-of-the-art *Classming* and *JavaTailor* by 27.9% and 18.6% in terms of the edge coverage on average. Meanwhile, we also demonstrate that all proposed mutators and the mutator scheduler are effective. Furthermore, *JITfuzz* successfully detects 36 previously unknown bugs none of which can be detected by *Classming* or *JavaTailor*. Specifically, 27 of them have been confirmed and 16 have been fixed by the corresponding developers. More specifically, 23 of them are JIT bugs, 18 have been confirmed, and 7 have been fixed.

VIII. ACKNOWLEDGEMENT

This work is partially supported by the National Natural Science Foundation of China (Grant No. 61902169), Guangdong Provincial Key Laboratory (Grant No. 2020B121201001), and Shenzhen Peacock Plan (Grant No. KQTD2016112514355531). This work is also partially supported by National Science Foundation under Grant Nos. CCF-2131943 and CCF-2141474, as well as Ant Group.

REFERENCES

[1] "List of jvm languages, wikipedia," https://en.wikipedia.org/wiki/List_of_JVM_languages, 2022.

[2] "Afl," https://lcamtuf.coredump.cx/afl/, 2022.

[3] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1032–1043. [Online]. Available: https://doi.org/10.1145/2976749.2978428

[4] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "Mopt: Optimized mutation scheduling for fuzzers," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, 2019, p. 1949–1966.

[5] C. Lemieux and K. Sen, *FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage*. New York, NY, USA: Association for Computing Machinery, 2018, p. 475–485. [Online]. Available: https://doi.org/10.1145/3238147.3238176

[6] "The java virtual machine specification," https://docs.oracle.com/javase/specs/index.html, 2022.

[7] Y. Chen, T. Su, and Z. Su, "Deep differential testing of jvm implementations," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 1257–1268.

[8] "How the jit compiler optimizes code," https://www.ibm.com/docs/en/sdk-java-technology/8?topic=compiler-how-jit-optimizes-code, 2022.

[9] Y. Zhao, Z. Wang, J. Chen, M. Liu, M. Wu, Y. Zhang, and L. Zhang, "History-driven test program synthesis for jvm testing," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1133–1144. [Online]. Available: https://doi.org/10.1145/3510003.3510059

[10] github, "Github," 2022. [Online]. Available: https://github.com/

[11] "Openjdk jdk 19 release-candidate builds," https://jdk.java.net/19/, 2022.

[12] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[13] "*JITfuzz*'s source code," https://github.com/lochnagarr/JITFuzz, 2022.

[14] "Openjdk," https://jdk.java.net/, 2022.

[15] "Openj9," https://www.eclipse.org/openj9/, 2022.

[16] "Oraclejdk," https://www.oracle.com/java/technologies/downloads/, 2022.

[17] "Jrockit," https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/webdocs/index.html, 2022.

[18] "Inlining," https://en.wikipedia.org/wiki/Inline_expansion, 2022.

[19] "Simplification," https://en.wikipedia.org/wiki/Computer_algebra#Simplification, 2022.

[20] "Escape," https://en.wikipedia.org/wiki/Escape_analysis, 2022.

[21] M. Paleczny, C. Vick, and C. Click, "The java hotspottm server compiler," in *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, ser. JVM'01. USA: USENIX Association, 2001, p. 1.

[22] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff, "Escape analysis for java," in *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 1–19. [Online]. Available: https://doi.org/10.1145/320384.320386

[23] K. Shiv, R. Iyer, C. Newburn, J. Dahlstedt, M. Lagergren, and O. Lindholm, "Impact of jit/jvm optimizations on java application performance," in *Seventh Workshop on Interaction Between Compilers and Computer Architectures, 2003. INTERACT-7 2003. Proceedings.*, 2003, pp. 5–13.

[24] "The jit compiler," https://www.ibm.com/docs/en/ztpf/1.1.0.15?topic=reference-jit-compiler, 2022.

[25] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," 06 2016, pp. 85–99.

[26] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 803–817.

[27] "Global escape," https://wiki.openjdk.org/display/HotSpot/EscapeAnalysis, 2022.

[28] R. Wilhelm, H. Seidl, and S. Hack, *Compiler design: syntactic and semantic analysis*. Springer Science & Business Media, 2013.

[29] Q. Tao, W. Wu, C. Zhao, and W. Shen, "An automatic testing approach for compiler based on metamorphic testing technique," in *2010 Asia Pacific Software Engineering Conference*. IEEE, 2010, pp. 270–279.

[30] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[31] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated testing of graphics shader compilers," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[32] "J9," http://www.ibm.com/developerworks/java/jdk, 2022.

[33] "Dragonwell8," https://github.com/alibaba/dragonwell8, 2022.

[34] "Dragonwell11," https://github.com/alibaba/dragonwell11, 2022.

[35] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser, "Program transformation with scoped dynamic rewrite rules," *Fundamenta Informaticae*, vol. 69, no. 1-2, pp. 123–178, 2006.

[36] "An example for depth-ensured transition," https://github.com/lochnagarr/JITFuzz/blob/main/examples/Depth-Ensured/NumberUtils.java.

[37] S. Hashima, M. M. Fouda, Z. M. Fadlullah, E. M. Mohamed, and K. Hatano, "Improved ucb-based energy-efficient channel selection in hybrid-band wireless communication," in *2021 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2021, pp. 1–6.

[38] X. Wang, J. Tang, M. Yu, G. Yin, and J. Li, "A ucb1-based online job dispatcher for heterogeneous mobile edge computing system," in *2018 Third International Conference on Security of Smart Cities, Industrial Control System and Communications (SSIC)*. IEEE, 2018, pp. 1–6.

[39] E. J. Powley, D. Whitehouse, and P. I. Cowling, "Bandits all the way down: Ucb1 as a simulation policy in monte carlo tree search," in *2013 IEEE Conference on Computational Inteligence in Games (CIG)*. IEEE, 2013, pp. 1–8.

[40] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer *et al.*, "The dacapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2006, pp. 169–190.

[41] T. McCabe, "A complexity measure," vol. SE-2, no. 4, 1976, pp. 308–320.

[42] L. Ardito, L. Barbato, M. Castelluccio, R. Coppola, C. Denizet, S. Ledru, and M. Valsesia, "rust-code-analysis: A rust library to analyze and extract maintainability information from source codes," *SoftwareX*, vol. 12, p. 100635, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711020303484

[43] Z. Gui, H. Shu, F. Kang, and X. Xiong, "Firmcorn: Vulnerability-oriented fuzzing of iot firmware via optimized virtual execution," *IEEE Access*, vol. 8, pp. 29 826–29 841, 2020.

[44] A. Calleja, J. Tapiador, and J. Caballero, "The malsource dataset: Quantifying complexity and code reuse in malware development," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 12, pp. 3175–3190, 2019.

[45] A. Kuleshov, P. Trifanov, V. Frolov, and G. Liang, "Diktat: Lightweight static analysis for kotlin," in *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2021, pp. 365–370.

[46] "hutool," https://github.com/dromara/hutool, 2022.

[47] "javapoet," https://github.com/square/javapoet, 2022.

[48] "mybatis-3," https://github.com/mybatis/mybatis-3, 2022.

[49] "zxing," https://github.com/zxing/zxing, 2022.

[50] "fastjson," https://github.com/alibaba/fastjson, 2022.

[51] "guice," https://github.com/google/guice, 2022.

[52] "commons-text," https://github.com/apache/commons-text, 2022.

[53] "rocketmq," https://github.com/apache/rocketmq, 2022.

[54] "spark," https://github.com/perwendel/spark, 2022.

[55] "vert.x," https://github.com/eclipse-vertx/vert.x, 2022.

[56] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang, and Y. Zhang, "One fuzzing strategy to rule them all," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022.

[57] L. Baresi and M. Pezze, "An introduction to software testing," *Electronic Notes in Theoretical Computer Science*, vol. 148, no. 1, pp. 89–111, 2006.

[58] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++: Combining incremental steps of fuzzing research," in *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*, 2020.

[59] D. She, R. Krishna, L. Yan, S. Jana, and B. Ray, "Mtfuzz: fuzzing with a multi-task neural network," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 737–749.

[60] T. W. MacFarland and J. M. Yates, "Mann–whitney u test," in *Introduction to nonparametric statistics for the biological sciences using R*. Springer, 2016, pp. 103–132.

[61] "Jdk-8280126," https://bugs.openjdk.org/browse/JDK-8280126, 2022.

[62] M. Paleczny, C. Vick, and C. Click, "The java HotSpot™ server compiler," in *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*. Monterey, CA: USENIX Association, Apr. 2001.

[63] "Openj9 optimizer vulnerability," https://github.com/eclipse-openj9/openj9/issues/15764, 2022.

[64] "Openj9 assertion failure," https://github.com/eclipse-openj9/openj9/issues/15639, 2022.

[65] "Fuzzing," https://en.wikipedia.org/wiki/Fuzzing, 2022.

[66] M. Wu, L. Jiang, J. Xiang, Y. Zhang, G. Yang, H. Ma, S. Nie, S. Wu, H. Cui, and L. Zhang, "Evaluating and improving neural program-smoothing-based fuzzing," in *Proceedings of the 44th International*

Conference on Software Engineering, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 847–858. [Online]. Available: https://doi.org/10.1145/3510003.3510089

[67] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang, and Y. Zhang, "One fuzzing strategy to rule them all," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1634–1645. [Online]. Available: https://doi.org/10.1145/3510003.3510174

[68] L. Jiang, H. Yuan, M. Wu, L. Zhang, and Y. Zhang, "Evaluating and improving hybrid fuzzing," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. New York, NY, USA: Association for Computing Machinery, 2023.

[69] Y. Noller and S. Tizpaz-Niari, "Qfuzz: Quantitative fuzzing for side channels," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 257–269. [Online]. Available: https://doi.org/10.1145/3460319.3464817

[70] A. Andronidis and C. Cadar, "Snapfuzz: High-throughput fuzzing of network applications," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 340–351. [Online]. Available: https://doi.org/10.1145/3533767.3534376

[71] M. Wu, Y. Ouyang, H. Zhou, L. Zhang, C. Liu, and Y. Zhang, "Simulee: Detecting cuda synchronization bugs via memory-access modeling," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 937–948. [Online]. Available: https://doi.org/10.1145/3377811.3380358

[72] M. Wu, L. Zhang, C. Liu, S. H. Tan, and Y. Zhang, "Automating cuda synchronization via program transformation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 748–759.

[73] Z. Ma, B. Zhao, L. Ren, Z. Li, S. Ma, X. Luo, and C. Zhang, "Printfuzz: Fuzzing linux drivers via automated virtual device simulation," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 404–416. [Online]. Available: https://doi.org/10.1145/3533767.3534226

[74] Y. Zheng, Y. Li, C. Zhang, H. Zhu, Y. Liu, and L. Sun, "Efficient greybox fuzzing of applications in linux-based iot devices via enhanced user-mode emulation," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 417–428. [Online]. Available: https://doi.org/10.1145/3533767.3534414

[75] S. Song, J. Hur, S. Kim, P. Rogers, and B. Lee, "R2z2: Detecting rendering regressions in web browsers through differential fuzz testing," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 1818–1829.

[76] W. Li, J. Shi, F. Li, J. Lin, W. Wang, and L. Guan, "afl: Non-intrusive feedback-driven fuzzing for microcontroller firmware," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–12. [Online]. Available: https://doi.org/10.1145/3510003.3510208

[77] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 132–142. [Online]. Available: https://doi.org/10.1145/3238147.3238187

[78] H. Zhou, W. Li, Z. Kong, J. Guo, Y. Zhang, B. Yu, L. Zhang, and C. Liu, "Deepbillboard: Systematic physical-world testing of autonomous driving systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 347–358. [Online]. Available: https://doi.org/10.1145/3377811.3380422

[79] T. Woodlief, S. Elbaum, and K. Sullivan, "Semantic image fuzzing of ai perception systems," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1958–1969. [Online]. Available: https://doi.org/10.1145/3510003.3510212

[80] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023, 2023.

[81] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "Deep learning library testing via effective model generation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 788–799.

[82] A. Wei, Y. Deng, C. Yang, and L. Zhang, "Free lunch for testing: Fuzzing deep-learning libraries from open source," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 995–1007.

[83] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 1027–1038.

[84] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "Nnsmith: Generating diverse and valid test cases for deep learning compilers," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 530–543. [Online]. Available: https://doi.org/10.1145/3575693.3575707

[85] S. Reddy, C. Lemieux, R. Padhye, and K. Sen, "Quickly generating diverse valid test inputs with reinforcement learning," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 1410–1421.

[86] M. Eberlein, Y. Noller, T. Vogel, and L. Grunske, "Evolutionary grammar-based fuzzing," in *Search-Based Software Engineering*, A. Aleti and A. Panichella, Eds. Cham: Springer International Publishing, 2020, pp. 105–120.

[87] T. Yoshikawa, K. Shimura, and T. Ozawa, "Random program generator for java jit compiler test system," in *Third International Conference on Quality Software, 2003. Proceedings.*, 2003, pp. 20–23.

[88] E. G. Sirer and B. N. Bershad, "Using production grammars in software testing," in *Proceedings of the 2nd Conference on Domain-Specific Languages*, ser. DSL '99. New York, NY, USA: Association for Computing Machinery, 2000, p. 1–13. [Online]. Available: https://doi.org/10.1145/331960.331965

[89] A. S. Boujarwah, K. Saleh, and J. Al-Dallal, "Testing syntax and semantic coverage of java language compilers," *Information and Software Technology*, vol. 41, no. 1, pp. 15–28, 1999.

[90] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 216–226. [Online]. Available: https://doi.org/10.1145/2594291.2594334

[91] Q. Zhang, C. Sun, and Z. Su, "Skeletal program enumeration for rigorous compiler testing," *SIGPLAN Not.*, vol. 52, no. 6, p. 347–361, jun 2017. [Online]. Available: https://doi.org/10.1145/3140587.3062379

[92] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," *SIGPLAN Not.*, vol. 51, no. 10, p. 849–863, oct 2016. [Online]. Available: https://doi.org/10.1145/3022671.2984038

[93] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing javascript engines with aspect-preserving mutation," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1629–1642.

[94] D. Huang, J.-B. Tristan, and G. Morrisett, "Compiling markov chain monte carlo algorithms for probabilistic modeling," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 111–125.

[95] C. Andrieu, N. De Freitas, A. Doucet, and M. I. Jordan, "An introduction to mcmc for machine learning," *Machine learning*, vol. 50, no. 1, pp. 5–43, 2003.