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

***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) is proposed to strengthen the efficacy of JVM. Therefore, how to effectively and efficiently detect JIT bugs becomes critical to ensure the correctness of JVM execution. In this paper, we propose a coverage-guided fuzzing framework, namely *JITfuzz*, to automatically detect JIT bugs. In particular, *JITfuzz* adopts a set of mechanism-activating mutators to facilitate the usage of the typical JIT optimization techniques, e.g., function inlining and simplification. Meanwhile, *JITfuzz* also adopts one basic-block-augmenting mutator and one transition-augmenting mutator to expand control flows of target programs. Moreover, *JITfuzz* also proposes a mutator scheduler which iteratively schedules mutators according to the coverage updates to strengthen the code coverage of JIT. To evaluate the effectiveness of *JITfuzz*, we conduct a set of experiments based on a benchmark with 10 popular JVM-based projects from GitHub. The experimental results suggest that *JITfuzz* outperforms the state of the art by 29.2% in terms of edge coverage on average. Furthermore, *JITfuzz* also successfully detects 32 bugs and 23 of them have been confirmed by the developers.**

I. Introduction

Since Java Virtual Machine (JVM) supports the executions of Java bytecode which can be compiled from multiple high-level programming languages, e.g., Java, Scala, and Clojure [1], it has been widely adopted in many popular application domains such as mobile applications, cloud computing, etc. However, while JVM is advanced in adopting interpretation in addition to compilation to realize cross-platform execution, interpreting the JVM-based programs can incur severe performance overhead due to the expensive loading stage and memory usage. To tackle this issue, the Just-in-Time compiler (JIT) is proposed to augment the runtime compilation performance of JVM-based applications by compiling the selected methods into native machine code. In this way, the resulting JVM bytecodes can be executed directly without the costly interpretation process and thus leads to efficient JVM execution. To date, JIT has become a crucial component in JVM where its correctness plays a vital role in ensuring correct and efficient execution of the 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 persistently tested because they include diverse optimization techniques which are activated under diverse scenarios. That indicates to thoroughly test JITs, it is essential for creating as many such optimization scenarios as possible, which can be potentially costly. 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 thus easily terminate 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 advance 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 breach the variable dependencies [7], causing early-terminated JVM executions, i.e., insufficient JIT testing, as well. Therefore, albeit JVM testing techniques, e.g., Classming [7] and ClassFuzz [8], can be used to occasionally expose JIT bugs, there still is a pressing need for sophisticated technique to specifically test JITs.

In this paper, we propose *JITfuzz*, a coverage-guided fuzzing framework to specifically test JITs. In particular, testing JITs essentially is equivalent to testing their mechanism of compiling bytecodes into native machine code at runtime. Intuitively, the application of such mechanism can be advanced when facilitating the usage of its optimization techniques and expanding the control flows of target JITs. Therefore in this paper, after explicitly launching JITs via specific JVM operations, *JITfuzz* adopts three mutator types to facilitate the application of the JIT mechanism. More specifically, mechanism-activating mutators are proposed to activate typical optimization techniques on purpose. For instance, the function-inlining-activating mutator is applied to create scenarios which facilitate the activation of the function inlining technique [9]. Meanwhile, noticing that mutating program control flows can potentially further augment the usage of optimization techniques, we also propose one basic-block-augmenting mutator and one transition-augmenting mutator to augment the usage of the program basic blocks and their transitions of the original program control flow. 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 composed of 10 popular open-source projects in Github [10], and then select one class with high cyclomatic complexity [11] from each project to form our benchmark suite. Next, by choosing OpenJDK19 [12] 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 JVM fuzzer *Classming* by 29.2% 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% on average. Moreover, *JITfuzz* successfully detects 32 previously unknown bugs on three commercial JVMs suite while none of them can be detected by *Classming*. In particular, 23 bugs have been confirmed by the corresponding developers and 12 have been fixed. More specifically, 19 of them are JIT bugs where 14 JIT bugs have been confirmed and 7 of them have been fixed.

In summary, our paper makes the following contributions:

- **Idea.** 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* [13] with the source code available in our Github page [14].
- **Evaluation.** We evaluate *JITfuzz* under multiple experimental setups, where the results mainly suggest that *JITfuzz* outperforms state-of-the-art *Classming* by 29.2% in terms of the code coverage. In addition, *JITfuzz* successfully detects 32 previously unknown bugs where 19 are JIT bugs.

II. Background

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

A. Coverage-guided Fuzzing

Fuzzing [15] refers to an automated software testing technique that inputs unexpected or random data to programs such that the program exceptions such as crashes, failing code assertions, or memory leaks can be exposed and monitored. Specifically, many fuzzers, e.g., AFL [2], MOPT [4], and Neuzz [16], adopt code coverage as guidance to facilitate bug/vulnerability exposure. More specifically, coverage-guided fuzzers generate mutants and retain a mutant as a seed for further mutations if it increases/optimizes code coverage of target programs. Although being proven effective, many coverage-guided fuzzers still are restricted by early-terminated execution on target programs which potentially results in limited testing coverage [8, 7] and further compromises the effectiveness of bug/vulnerability exposure. For instance, conventional JVM fuzzers *Classming* [7] and *ClassFuzz* [8] tend to be restricted in exploring deep execution states for target JVMs since running many of their resulting mutants can fail program executions early, e.g., in the verification phase. Consequently, such mutants cannot be used for testing the deep states of JVMs, e.g., their execution engines. To alleviate such issues, intuitively, one can refine the mutator designs and scheduling strategies for exploring deep execution states of target JVMs, e.g., JITs.

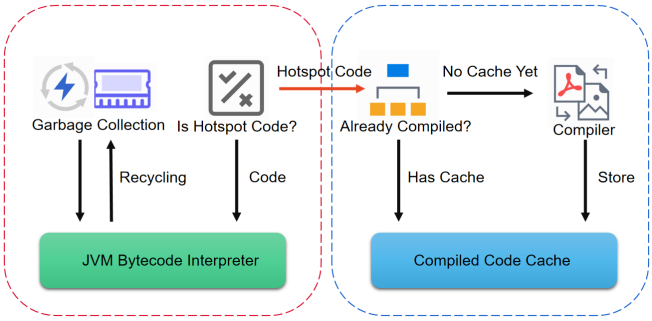

Fig. 1: The framework of JVM

B. JVM JIT

In JVM, the Just-in-time compiler (JIT) selects "Hotspot" methods running on JVM and compiles such methods 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 method is "Hotspot" [17] (i.e., 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 "Hotspot" 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 [9] to optimize program compilation performance at runtime for realizing efficient JVM execution via control flow analysis. Specifically, many widely-used JVMs, e.g., OpenJDK [18], OpenJ9 [19], OracleJDK [20], and JRockit [21], adopt common optimization techniques such as function inlining [22], simplification [23], escape analysis [24], and scalar replacement [25]. 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) [26]. 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.

C. Motivation

JIT has been demonstrated to significantly impact the runtime performance of JVM-based applications [27] and thus strongly recommended by industrial developers [28]. While testing JITs to ensure their correctness is essential, there exists no testing technique for such specific purpose

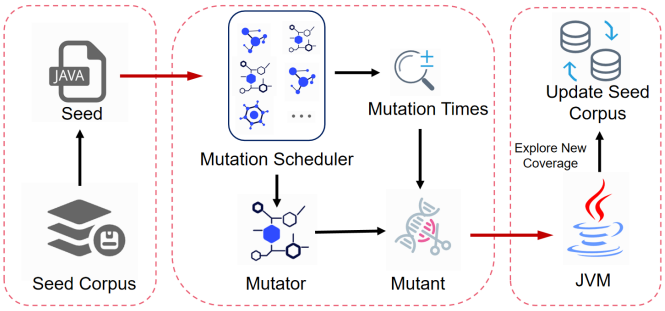

Fig. 2: The framework of *JITfuzz* Yuqun:[get rid of mutation times, place multiple mutators with "mutator limit" in the label. how do we deal with mutant which cannot explore edges?]

to our best knowledge. Albeit the existing general-purpose JVM testing techniques [8, 7] can potentially expose JIT bugs occasionally, they encounter severe challenges to prevent them from effectively exposing the JIT bugs. In particular, when *ClassFuzz* [8] randomly manipulates (e.g., deletes or inserts) instructions and *Classming* [7] intentionally breaches the variable dependencies to expose erroneous data flows, they potentially cause early-terminated JVM executions which can render the JIT optimization techniques insufficiently activated, and thus result in insufficient testing of JITs. 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 mutate control flows while preventing early termination of their associated testing runs.

III. Approach

We propose *JITfuzz*, a coverage-guided fuzzer for JVM JIT with three mutator types and a mutation scheduler. In particular, *JITfuzz* is implemented with Jimple-level instructions provided by *Soot* [13] which is a framework for analyzing and transforming JVM-based applications. Figure 2 presents the overall workflow of *JITfuzz*. Typically, Initialized with a *seed corpora*, *JITfuzz* iterates each *seed* to generate mutants under a given time budget. For each iteration, *JITfuzz* first 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 mechanism-activating mutators, one basic-block-augmenting mutator, and one transition-augmenting mutator to facilitate the usage of JIT optimization techniques and expand the control flows. Accordingly, *JITfuzz* adopts a mutator scheduler to select and schedule the mutators of 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 corpora* 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 mechanism-activating mutators. Specifically, we first analyze typical JIT optimization techniques, e.g., simplification [23] and escape analysis [26, 29], and then derive the strategies which aim for extensively applying the corresponding optimization techniques. As a result, such strategies are adopted as the mechanism-activating mutators.

We further realize that mutating control flows in JIT can potentially advance the the effectiveness of JIT fuzzing due to the following reaonss. First, control-flow analysis can potentially advance the JIT optimization techniques by identifying where and how to optimize JVM-based programs [9]. Second, control-flow analysis can facilitate the correct compilation from Java programs into native machine codes by verifying correctness of semantics [30], etc. At last, existing works [31, 32, 33, 7] also demonstrate that diversifying control flows in running programs can significantly increase fuzzing performance, e.g., coverage. Accordingly in *JITfuzz*, we propose a basic-block-augmenting mutator based on control-flow analysis to augment the usage of *basic block*s [34] (nodes in control flow graphs) for a given seed, and the transition-augmenting mutator to augment the usage of transitions among *basic block*s (edges in control flow graphs) respectively. 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 operation option `java -Xcomp cls` for a given class `cls`.

1) Mechanism-activating Mutators: While mechanism-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 types of mechanism-activating mutators corresponding to the most commonly adopted corresponding optimization techniques in the existing JITs for simplicity. i.e., function inlining [22], simplification [23], escape analysis [24], and scalar replacement [25] which are commonly adopted by the JITs from diverse well-recognized JVMs [18, 35, 36, 37]. Table I shows the small-step illustrations [38] for the details of the mechanism-activating mutators. Specifically, 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. Specifically, given an expression $\alpha\ op\ \beta$ (op denotes a binary operator) with the execution state σ, we create a new function $\rho(x, y)$ which returns the expression $x\ op\ y$ with the updated execution state σ'. Consequently,we mutate the original expression $\gamma = \alpha\ op\ \beta$ as its corresponding transformation $\gamma = \rho(\alpha, \beta)$. 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. Next, noticing that simplification essentially refers to simplifying an arithmetic expression, the simplification-activating mutator replaces a simplistic arithmetic expression as a semantic-preserving yet complicated expression. As stated in Rule 2, we update the original expression α with its semantics-preserving expression $\alpha + 0$ and generate an expression $expr$ which is calculated to be zero. As a result, we update the original expression $\gamma = \alpha$ as $\gamma = \alpha + expr$. 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 in order to facilitate the application of simplification on the expression. Furthermore, 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 α *op* β. Example 3 shows that we create a `Digit` object `r0` to mutate the constant integer `0` in the original instruction with 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 [26] when JIT verifies whether an object in the heap has side effects or not. At last, to facilitate the escape analysis in the *GlobalEscape* level [26], 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 copy the local object by `i0.deepcopy()` 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) Basic-Block-Augmenting Mutator: Intuitively, to increase the number of *basic block*s in the existing control flows without devastating their executions, one can design an *if(true)* expression and/or a *loop(limit)* expression to contain the existing program statements. Accordingly, our basic-block-augmenting mutator is proposed to contain a given instruction within *if* and/or *loop* statement(s). Algorithm 1 presents the details of how an basic-block-augmenting mutator is applied under an input instruction $\mathcal{I}$, its associated *seed*, and an 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*

Algorithm 1 Basic-Block-Augmenting 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)
```

(line 9) followed by updating *Cnt* (line 10). Note that all the above process is recursively executed (line 11).

3) Transition-Augmenting Mutator: We also propose transition-augmenting mutators to augment the transitions between basic blocks. While we simply select two random basic blocks for applying the transition-augmenting mutator to generate their transition, we also realize that only increasing basic block transitions without ensuring the correct execution of the associated target programs 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 (the code and its corresponding labels can be found in [39]) where the dark solid lines denote the existing transitions between basic blocks. Assume by applying the basic-block-augmenting mutators, a transition is established from `f8` to `f5` and thus leads to an execution path `[f1, f7, f8, f5]`. However, since `f5` depends on variables `sampleIndex` and `samplePos` [39] 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 prevent further testing on the deep states of JIT, e.g., the optimization techniques. Therefore, when designing the transition-augmenting mutator, we also need to resolve the potential dependency issues so as to facilitate testing deep states of target programs.

Concepts. We start our illustration of the transition-augmenting mutator by defining the following concepts.

- *ENTRY.* A *basic block* in a seed program which is executed to initialize JVM execution.
- *Ignored edge*. A transition between *basic block*s created by executing a *loop* expression or applying a transition-augmenting mutator.
- *Directed CFG.* A directed control-flow graph generated by deleting all *ignored edge*s from the orginal control-flow graph (CFG) of a seed program.
- *Dominator sequence.* A topologically reversed-ordered dominator [40] list of a *basic block* derived from its corresponding *directed CFG*.

We take Figure 3 as an example to illustrate all these definitions. In particular, `f1` is the *ENTRY*. We then recognize the transitions `f3`→`f2` and `f8`→`f2` as *ignored edge*s since they are respectively created by a *loop* expression and

TABLE I: Mechanism-triggered Mutators

Mechanism	Rules for mechanism-activating mutators	Example

Function Inlining	**Rule 1:** $\frac{\langle \alpha\ op\ \beta, \sigma\rangle \rightarrow \langle \rho(x,y) : \textbf{return}\ x\ op\ y, \sigma^{'}\rangle}{\langle \gamma = \alpha\ op\ \beta, \sigma\rangle \rightarrow \langle \gamma = \rho(\alpha,\beta), \sigma^{'}\rangle}$	**Example 1:**
		`-int i2 = i0 + i1;`
		`+public int inline(int i0, int i1) {`
		`+  return i0 + i1;`
		`+}`
		`+int i2 = inline(i0, i1);`

Simplification	**Rule 2:** $\frac{\langle \alpha\ , \sigma\rangle \rightarrow \langle \alpha + 0, \sigma^{'}\rangle, \langle 0, \sigma\rangle \rightarrow \langle expr, \sigma^{'}\rangle}{\langle \gamma = \alpha, \sigma\rangle \rightarrow \langle \gamma = \alpha + expr, \sigma^{'}\rangle}$	**Example 2:**
		`-int i2 = i0 + i1;`
		`+int i3 = new Random().nextInt();`
		`+int i2 = (i0 + i3) + (i1 - i3);`

Scalar Replacement & Escape Analysis	**Rule 3:** $\frac{\langle \alpha\ , \sigma\rangle \rightarrow \langle obj[\![field]\!] = \alpha, \sigma^{'}\rangle}{\langle \gamma = \alpha\ op\ \beta, \sigma\rangle \rightarrow \langle \gamma = obj[\![field]\!]\ op\ \beta, \sigma^{'}\rangle}$	**Example 3:**
		`-int i0 = 0;`
		`+Digit r0 = new Digit(0);`
		`+int i0 = r0.integer;`

Escape Analysis	**Rule 4:** $\frac{\langle \alpha\ , \sigma\rangle \rightarrow \langle this[\![field]\!] = \alpha, \sigma^{'}\rangle}{\langle \gamma = \alpha\ op\ \beta, \sigma\rangle \rightarrow \langle \gamma = this[\![field]\!]\ op\ \beta, \sigma^{'}\rangle}$	**Example 4:**
		`-Object i0 = new Object();`
		`+this.object = i0.deepcopy();`
		`+Object i0 = this.object;`

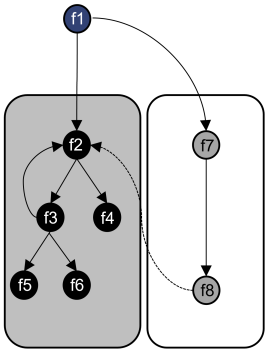

Fig. 3: An example [39] for transition-augmenting Yuqun:[should have a transition from f8 to f5 in different shape]

the transition-augmenting mutator, which will be illustrated laterYuqun:[did we?]. After removing them, we obtain a *directed CFG*. At last, the resulting *dominator sequence* of `f5` is [`f3`, `f2`, `f1`].

Algorithm. Note that according to *Soot* [13], 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 block*s (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 *ENTRY* among diverse seeds, their remaining partial control flows can be quite alike or even identical, causing limited program execution spaces and thus hindering program state explorations. Therefore in this paper, for a randomly generated basic block transition $\alpha \rightarrow \beta$, we determine to redirect it from α to the first dominator in the *dominator sequence* of β to prevent undefined variable errors when applying the transition-augmenting mutator.

Algorithm 2 illustrates the details our transition redirection mechanism when applying the transition-augmenting mutators. Given an input seed, we first construct its control-flow graph

Algorithm 2 Transition Redirection Mechanism

Input: *seed*
Output: *mutant*

```
1:  function RedirectBasicBlockTransition
2:      controlFlowGraph ← obtainCFG(seed)
3:      ENTRY ← identifyEntry(seed)
4:      directedCFG ← deleteIgnorededges(controlFlowGraph)
5:      src ← randomSelectBasicBlock(directedCFG)
6:      sink ← randomSelectBasicBlock(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 dominator before sinkFiC from srcPre
19:     mutant ← create a transition from src to sinkNew
20:     return mutant
```

and identify its *ENTRY* (lines 2 to 3), and then generate a *directed CFG* by deleting all *ignored edge*s (line 4). After randomly selecting the source *basic block* `src` and the sink *basic block* `sink` (lines 5 to 6), we derive the *dominator sequence* of `src` and store it in the set `srcPre` (lines 9 to 11). Next, we identify the *dominator sequence* of `sink` and store it in the set `sinkPre`. Accordingly, we store the first common dominator from `sinkPre` and `srcPre` as `sinkFiC` (lines 12 to 17). Finally, we identify the predecessor ahead of `sinkFiC` in the set `srcPre` as `sinkNew` and create a transition from `src` to `sinkNew` to generate a mutant (lines 18 to 19). For example in Figure 3, assume `f8` is `src` and `f5` is `sink`. Next, we obtain the first common dominator for `f8` and `f5`, i.e., `f1`. Then, `f2` is identified as the predecessor of `f1` in the *dominator sequence* of `f5`. At last, we create transition `f8`→`f2` to generate a mutant.

Proof. We then prove that Algorithm 2 is capable of prevent-

ing early termination of program executions and limited program state exploration. In particular, it is equivalent to proving that the sink dominator `sinkNew` selected by Algorithm 2 is the first dominator in the *dominator sequence* of `sink` causing no undefined variable error. We first prove that redirecting the transition to `sinkNew` cannot incur any undefined variable error via proof of contradiction. Specifically, assume that redirecting the transition from `src` to `sinkNew` incurs an undefined variable error out of the original error-free program execution, i.e., a variable is used without a definition in `sinkNew` after redirection. Then we infer that such variable can also be accessed by `sinkFiC` since `sinkFiC` dominates `sinkNew` in `srcPre`. Accordingly, we infer that such undefined variable error can be spread in `sinkFiC` (otherwise, `sinkFiC` is supposed to define variables to prevent the undefined variable error in `sinkNew`). However, such inference contradicts the assumption that the original program execution incurs no errors. Thus, redirecting the transition to `sinkNew` can be ensured to incur no undefined variable error. Next, we prove that `sinkNew` is the first dominator which the transition can be redirected to in the *dominator sequence* of `sink` without causing any undefined variable error. Similarly, assume there is a predecessor of `sinkNew` in the *dominator sequence* of `sink` (i.e., dominated by `sinkNew`) which the transition can be redirected to without causing any undefined variable error. Since redirecting the transition from `src` to such predecessor 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 such predecessor of `sinkNew`). Such inference contradicts the fact that `sinkNew` is allowed to define new variables as a dominator of the *dominator sequence* of `sink`. Q.E.D.

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 thir 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 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 [41], a lightweight algorithm which constructs an optimistic guess to the expected payoff of each action and pick 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 [41, 42, 43]. 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, under the scheduled mutation limit, *JITfuzz* iteratively selects each individual mutator out of all the six possible mutator options (i.e., four mechanism-activating mutators, one basic-block-augmenting mutator, and one transition-augmenting mutator). More specifically, the mutators can be scheduled via Equation 1 where $result(t)$ denotes both the mutation limit result and each corresponding individual 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.

IV. Evaluation

In this section, we conduct a set of experiments to evaluate the effectiveness and efficiency of *JITfuzz*. In particular, we first construct a real-world benchmark suite with 10 popular open-source Java projects in Github. Next, we compare *JITfuzz* with state-of-the-art *Classming* in terms of the edge coverage results obtained from JIT of *reference 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?*

Moreover, we report and analyze the bugs on our adopted benchmark exposed by *JITfuzz* with all the evaluation details presented in our GitHub page [14].

A. Benchmark Construction

In this paper, we define a set of rules to collect important and influential real-world Java projects in GitHub to form our benchmark for our evaluation. In particular, we first search

the keyword "Java" in GitHub and collect more than 5,000 projects. Next, we randomly select 10 projects with large star number (larger than 100) and LoC numbers (larger than 10k). As a result, we adopt "hutoo" [44], "javapoet" [45], "mybatis-3" [46], "zxing" [47], "fastjson" [48], "guice" [49], "commons-text" [50], "rocketmq" [51], "spark" [52] and "vert.x" [53] to constitute our benchmark with their detailed information presented in Table II. Furthermore, for each of these projects, we randomly select one of the ten classes with the highest cyclomatic complexity [11] as the seed program where the cyclomatic complexity refers to the number of linearly independent paths through the source code of a class [54, 55, 56, 57].

TABLE II: Benchmark

Project	Stars	LoC	Initial class(seed)
hutool	23.5k	265.7k	..core.text.PasswdStrength.class
javapoet	9.7k	12.4k	..ClassName.class
mybatis-3	17.5k	161.3k	..ibatis.parsing.GenericTokenParser.class
zxing	29.9k	219.3k	..zxing.qrcode.encoder.Encoder.class
fastjson	24.8k	103.9k	..fastjson.JSON.class
guice	11.3k	110.6k	..inject.spi.InjectionPoint.class
commons-text	242	54.7k	..commons.text.numbers.ParsedDecimal.class
rocketmq	17.8k	178.2k	..rocketmq.filter.util.BloomFilter.class
spark	9.3k	23.1k	spark.resource.UriPath.class
vert.x	3.3k	215.4k	..vertx.core.json.JsonArray.class

B. Environment Setup

We perform our evaluations on a desktop machine, 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) [18] 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 [14] due to page limit. Moreover, following many prior work [2, 16, 58], we adopt the edge coverage [59] to reflect the effectiveness of our studied techniques.

We select state-of-the-art *Classming* for performance comparison. Specifically, all our experiments are run for 24 hours following prior work [58, 16, 60]. Note that we run each experiment five times for obtaining the average results to reduce the impact of randomness.

C. Result Analysis

TABLE III: Mutant generation and edge coverage

Benchmark	*JITfuzz*		*Classming*	
	Mutant	Coverage	Mutant	Coverage
hutool	5,579	37,006	6,468	32,994
javapoet	10,400	36,178	8,247	33,948
mybatis-3	10,689	29,209	9,252	18,708
zxing	21,739	36,184	4,285	28,350
fastjson	14,580	36,222	7,480	26,121
guice	3,929	29,805	11,578	30,757
commons-text	12,443	36,863	4,615	34,213
rocketmq	12,870	32,808	11,476	29,043
spark	7,491	33,936	21,520	12,130
vert.x	12,185	34,056	3,840	24,162
Average	**11,191**	**34,931**	**8,876**	**27,042**

1) RQ1: the effectiveness of JITfuzz: The evaluation results of *JITfuzz* and *Classming* are presented in Table III where `Mutant` denotes the number of mutants generated and `Coverage` refers to the edge coverage results obtained by the studied fuzzers.

Overall, we observe that *JITfuzz* can significantly outperform *Classming* in terms of the edge coverage. Specifically, *JITfuzz* can explore averagely 34,931 edges, while *Classming* only explores 27,042 edges, i.e., *JITfuzz* explores over 29.2% more edges than *Classming*. Moreover, we further find that *JITfuzz* can generate mutants more efficiently than *Classming* by 26% (11,191 vs. 8,876 mutants within 24 hours) on average. Nevertheless, although in certain benchmarks (i.e., `hutool`, `guice` and `spark`), *Classming* generates averagely 0.33× more mutants (13,188 vs. 5,666 mutants)Yuqun:[wrong number] compared to *JITfuzz* in these benchmarks, *JITfuzz* still outperforms *Classming* in such benchmarks by exploring 32.8% more edges (33,582 vs. 25,294 explored edges) on average. Such results can further reflect that *JITfuzz* with its mutators and mutator scheduler can be quite effective compared to state of the art.

> *Finding 1: JITfuzz is more effective and efficient than Classming by exploring 29.2% more edges and generating 26% more mutants on average.*

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

Effectiveness of the mutators. First, we investigate the effectiveness of each mutator respectively. In particular, 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 $JITfuzz_{-escape}$, $JITfuzz_{-block}$, 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 basic-block-augmenting mutator, and the transition-augmenting mutator. Accordingly, we can derive the effectiveness of a mutator by comparing the performance of its associated technique variant with the original *JITfuzz*. Table IV demonstrates the edge coverage results of *JITfuzz* and our studied technique variants. In general, we can observe that *JITfuzz* outperforms each variant averagely from 11.8% to 23.7% in terms of edge coverage, which is rather substantial. Moreover, we also find that each variant can still outperform *Classming* in terms of edge coverage from 4.4% to 15.5% respectively, 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.*

TABLE IV: Effectiveness of the *JITfuzz* mutators

Benchmark	*JITfuzz*	*Classming*	Technical Variants by Disabling Mutators						Other Variants	
			$JITfuzz_{-scalar}$	$JITfuzz_{-escape}$	$JITfuzz_{-simp}$	$JITfuzz_{-inline}$	$JITfuzz_{-block}$	$JITfuzz_{-trans}$	$JITfuzz_{rand}$	$JITfuzz_{randtr}$
hutool	37,006	32,994	34,209	34,483	35,276	32,018	29,378	30,374	35,038	31,367
javapoet	36,178	33,948	33,675	30,990	34,296	34,135	32,381	32,070	33,599	32,465
mybatis	29,209	18,708	23,165	22,354	25,358	24,126	22,874	24,642	26,064	16,560
zxing	36,184	28,350	35,214	30,161	34,931	35,680	34,474	28,731	32,156	29,682
fastjson	36,222	26,121	30,114	34,367	34,963	33,715	33,269	28,711	33,874	26,341
guice	36,852	30,757	29,805	31,241	30,571	24,521	26,937	28,227	32,144	28,227
commons-text	36,863	34,213	35,075	36,075	36,881	34,503	32,958	34,039	33,039	33,129
rocketmq	32,808	29,043	26,977	28,457	24,097	26,241	27,322	24,060	30,422	25,638
spark	33,936	12,130	28,145	24,270	24,854	32,568	28,613	26,279	29,895	25,638
vert.x	34,056	24,162	28,056	23,806	31,199	22,785	27,322	25,302	32,438	25,986
average	**34,931**	**27,042**	**30,443**	**29,620**	**31,242**	**30,029**	**29,552**	**28,243**	**31,866**	**27,503**

Effectiveness of the mutator scheduler. To investigate the effectiveness of mutator scheduler, we build a technique variant $JITfuzz_{rand}$, which randomly schedules the mutators and the mutation limit in each iterative execution instead of applying the mutator scheduler.

In general, we can observe from Table IV that mutator scheduler is effective since *JITfuzz* outperforms $JITfuzz_{rand}$ significantly by 10% (34,931 vs. 31,866 explored edges) on average in terms of edge coverage. More specifically, we can further observe that *JITfuzz* outperforms $JITfuzz_{rand}$ 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 3: Adopting mutator scheduler can significantly improve the power of exploring new edges for JITfuzz by scheduling mutators and its mutation limit.*

Effectiveness of the transition redirection mechanism. We further investigate the effectiveness of the transition redirection mechanism for applying the transition-augmenting mutator. As in Algorithm 2, the transition-augmenting 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 block*s without applying any redirection.

We can observe from Table IV *JITfuzz* significantly outperforms $JITfuzz_{randtr}$ by 27% on average in terms of edge coverage, which demonstrates that redirecting transition is essential to facilitate the efficacy of the transition-augmenting mutator.

> *Finding 4: Transition redirection mechanism is essential for the transition-augmenting mutator in augmenting its edge coverage performance.*

D. Bug Report and Analysis

JITfuzz is effective in exposing multiple real-world JIT/JVM bugs. Note that a bug is defined as a defect related to a specific JVM version in this section. Specifically, we apply the seeds generated by *JITfuzz* in our evaluation to run multiple JVMs, i.e., different versions of OpenJ9 [19], OpenJDK [18], and OracleJDK [20], for exposing their bugs. Table V demonstrates the detailed information where we have successfully detected 32 JVM bugs. After reporting them to the corresponding JVM developers, 23 of them have been confirmed and 12 of them have been fixed. More specifically, 19 of them are JIT bugs, 14 have been confirmed, and 7 are fixed by the developers. Note that none of the bugs can be detected by *Classming*. We then illustrate three typical bugs exposed by *JITfuzz* as follows.

TABLE V: 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	0	8	0	4	0	0
TOTAL	19	13	14	9	7	5

1) JIT segmentation fault: We have reported a vulnerability [61] on the C2 compiler Yuqun:[citation]—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 by developers. This vulnerability is exposed by running the original JUnit tests with a generated class from `JSON.class`. In particular, the affected JVMs crashed due to a segment fault occurred within `ciMethodBlocks::make_block_at(int)`, as shown in Figure 4.

```
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.

The developers located the issue to two methods in the generated class whose bytecodes end abruptly with unreachable basic blocks, as shown in Figure 5. They found that HotSpot builds control flow graphs for unreachable *basic block*s where JIT fails to validate. As a result, by compiling unreachable *basic block*s, JIT accesses invalid memory, causing the seg-

```
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.

mentation fault in the C2 compiler. 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."*

2) Dead loop assertion failure: We discovered a HotSpot vulnerability on JIT caused by an assertion failure, which indicates that a dead loop was detected, as shown 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 was assigned with a bug ID JDK-8280126.

```
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.

The developers 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 take unexpected data as input and cause a dead loop, i.e., Yuqun:[or 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) Other runtime vulnerabilities: In addition to JIT vulnerabilities, *JITfuzz* is able to reveal runtime defects. For example, we have reported an OpenJ9 vulnerability 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 7.

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

```

Fig. 7: Assertion failure during verification

The developers further found that OpenJ9 incorrectly built stackmaps for the class file due to its huge size. More specifically, OpenJ9 represents the stack size and number of local variables within a method with a 16-bit unsigned integer. Consequently, for a method with much more than 65535 local variables, OpenJ9 cannot allocate the memory for its locals and stack and the crash occurred afterwards when it tries to release the memory of the stackmap frame:

> *"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 in which both locals and stack are empty.

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 randomly select 10 projects with large star/LoC numbers from 5,000 collected GitHub projects, then randomly select one class with high cyclomatic complexity from each project as the seed.

The threats to internal validity mainly lie in the potential flaws in our implementation of *JITfuzz*. To reduce the threats, 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.

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 quite detailed bug report from real-world benchmarks to strengthen the evaluation on the real-world applicability of *JITfuzz*.

VI. RELATED WORK

A. Fuzzing

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 identify branches exercised by few AFL-produced seeds, then invested 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. Furthermore, the core idea of fuzzing can be adapted to multiple scenarios. For example, Noller et al. [62] 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. [63] leveraged the power of snapshot of net applications to facilitate the fuzzing efficacy of networking applications. Ma et al. [64] proposed PrIntFuzz to fuzz Linux drivers by a contructed device simulator. Zheng et al. [65] proposed EQUAFL to fuzz Linux-based IoT devices via switching emulation types. Woodlief et al. [66] developed semSensFuzz to fuzz the AI perception systems via semantic mutations on the image inputs. By using raw html files as inputs, Song et al. [67] proposed R2Z2 which utilizes differential testing framework to fuzz web browsers. With the help of the debugging feature, Li et al. [68] proposed μAFL to bridge the fuzzing environment on PC and target firmware on microcontroller devices.

Although many existing fuzzers have been proven effective in fuzzing real-world programs, there exists no fuzzer specifically for JITs. In this paper, we propose *JITfuzz*, the first coverage-guided JIT fuzzer which designs multiple effective mutators and one 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 systematically parse existing real-world codes for producing discrepancy-induced programs, Le et al. [69] introduced an applicable methodology equivalence modulo inputs (EMI) to validate the quality of compilers. Researchers also focus on developing approaches to automatically generate programs for compiler testing. Yang et al. [32] 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. [70] proposed RLCheck, which utilizes reinforcement learning to generate diverse valid inputs to explore the programs which requires strict validation on the inputs, e.g., JavaScript engine. Eberlein et al. [71] 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). To test JVM, Yoshikawa et al. [72] proposed a random Java program generator based on the predefined syntax to expose vulnerabilities in JVM by differential testing. Sirer et al. [73] proposed lava to generate Java programs for JVM testing via randomly iterating over the Java grammar productions. Boujarwah et al. [74] utilized the predefined grammar of Java programs to generate semantics-correct test cases to fuzz JVM. Zhao et al. [75] proposed JavaTailor, which generates new 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., ZestYuqun:[citation]), or require massive program samples with additional efforts (e.g., CsmithYuqun:[citation]), the mutation-based testing approaches are usually launched with specifically designed mutators and only one seeding program. Zhang et al. [76] introduced the concepts of the skeletal program enumeration, then replaced the variables in the original skeletal program to generate new control flows for compiler testing. By removing the restriction of only mutating the unreachable regions of the input programs, Sun et al. [77] tested compilers via mutating the live code of the input programs. Donaldson et al. [33] developed GLFuzz, for testing OpenGL shading language compilers based on semantics-preserving program transformations. Park et al. [78] proposed Die to fuzz the JavaScript engine via the aspect preservation mutators. In terms of testing JVM, Chen et al. [8] proposed classfuzz, which utilizes Markov Chain Monte CarloYuqun:[citation] to guide the 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 using only one given seed and explores JIT compiler thoroughly with the well-designed mutators and mutation scheduler.

VII. CONCLUSION

In this paper, we develop a coverage-guided fuzzing framework for JVM JITs, namely *JITfuzz* which includes four mechanism-activating mutators, one basic-block-augmenting mutator, and one transition-augmenting mutator. Moreover, *JITfuzz* adopts a lightweight mutator scheduler to augment the overall effectiveness of applying the mutators. To evaluate the effectiveness of *JITfuzz*, we construct a benchmark with 10 seeds from real-world JVM-based projects. Our evaluation results suggest that *JITfuzz* can outperform state-of-the-art *Classming* by 29.2% in terms of the edge coverage on average. Furthermore, *JITfuzz* successfully detects 32 previously unknown bugs where 23 of them have been confirmed and 12 have been fixed by the corresponding developers. More specifically, 19 of them are JIT bugs, 14 have been confirmed, and 7 have been fixed.

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

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

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

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

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

[13] 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.

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

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

[16] 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.

[17] "The definition of "hotspot" method in jit," https://docs.oracle.com/javacomponents/jrockit-hotspot/migration-guide/comp-opt.htm#JRHMG142, 2022.

[18] "openjdk," https://jdk.java.net/, 2022.

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

[20] "oraclejdk," https://www.oracle.com/java/technologies/downloads/, 2022.

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

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

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

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

[25] 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.

[26] 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

[27] 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.

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

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

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

[31] 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.

[32] 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.

[33] 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.

[34] "Basic block, wikipedia," https://en.wikipedia.org/wiki/Basic_block, 2022.

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

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

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

[38] M. Fernández, "Programming languages and operational semantics: a concise overview," 2014.

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

[40] "Dominator and immediate dominator, wikipedia," https://en.wikipedia.org/wiki/Dominator_(graph_theory), 2022.

[41] 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.

[42] 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.

[43] 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.

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

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

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

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

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

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

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

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

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

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

[54] 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

[55] 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.

[56] 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.

[57]

[58] 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.

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

[60] 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.

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

[62] 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

[63] 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

[64] 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

[65] 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

[66] 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

[67] 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.

[68] 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

[69] 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

[70] 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.

[71] 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.

[72] 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.

[73] 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

[74] 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.

[75] 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

[76] 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

[77] 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

[78] 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.