# *SJFuzz*: Seed and Mutator Scheduling for JVM Fuzzing


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

Yicheng Ouyang†
Minghai Lu
11610313@mail.sustech.edu.cn
11910620@mail.sustech.edu.cn
Southern University of Science and Technology
Shenzhen, China

Junjie Chen
Yingquan Zhao
junjiechen@tju.edu.cn
zhaoyingquan@tju.edu.cn
College of Intelligence and Computing, Tianjin University
Tianjin, China

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

Guowei Yang
The University of Queensland
Queensland, Australia
guowei.yang@uq.edu.au

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



## ABSTRACT

While the Java Virtual Machine (JVM) plays a vital role in ensuring correct executions of Java applications, testing JVMs via generating and running class files on them can be rather challenging. The existing techniques, e.g., *ClassFuzz* and *Classming*, attempt to leverage the power of fuzzing and differential testing to cope with JVM intricacies by exposing discrepant execution results among different JVMs, i.e., inter-JVM discrepancies, for testing analytics. However, their adopted fuzzers are insufficiently guided since they include no well-designed seed and mutator scheduling mechanisms, leading to inefficient differential testing. To address such issues, in this paper, we propose *SJFuzz*, the first JVM fuzzing framework with seed and mutator scheduling mechanisms for automated JVM differential testing. Overall, *SJFuzz* aims to mutate class files via control flow mutators to facilitate the exposure of inter-JVM discrepancies. To this end, *SJFuzz* schedules seeds (class files) for mutations based on the discrepancy and diversity guidance. *SJFuzz* also schedules mutators for diversifying class file generation. To evaluate *SJFuzz*, we conduct an extensive study on multiple representative real-world JVMs, and the experimental results show that *SJFuzz* significantly outperforms the SOTA mutation-based and generation-based JVM fuzzers in terms of the inter-JVM discrepancy exposure. Moreover, *SJFuzz* successfully reported 46 potential JVM issues where 20 were confirmed as bugs and 16 have been fixed by the JVM developers.



*Mingyuan Wu is also affiliated with the Research Institute of Trustworthy Autonomous Systems, Shenzhen, China.
†Yicheng Ouyang is now a PhD student in University of Illinois Urbana-Champaign.
‡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




*ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA*



ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616277



## CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

## KEYWORDS

JVM Testing, Mutation-based Fuzzing

**ACM Reference Format:**
Mingyuan Wu, Yicheng Ouyang, Minghai Lu, Junjie Chen, Yingquan Zhao, Heming Cui, Guowei Yang, and Yuqun Zhang. 2023. *SJFuzz*: Seed and Mutator Scheduling for JVM Fuzzing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616277


## 1 INTRODUCTION

The Java Virtual Machine (JVM) refers to the virtual machine which interprets and executes Java bytecode compiled from various high-level programming languages, e.g., Java, Scala, and Clojure [20]. Typically, after source code files are compiled to bytecode class files, JVM first leverages class loaders to load such class files, in terms of the strict order of loading, linking, and initialization. Then, JVM directly executes the bytecode, or transforms the loaded bytecode into machine code for actual execution via Just-in-Time (JIT) or Ahead-of-Time (AOT) compilers for optimization purposes.

Multiple JVM implementations, such as Oracle's HotSpot [15], Alibaba's DragonWell [11, 12], IBM's OpenJ9 [16], Azul's Zulu [23], and GNU's GIJ [14], have been widely applied in support of a variety of Java-bytecode-based applications. While ideally they are expected to implement the same JVM specification and conform to consistent cross-platform robustness, they are usually implemented by different groups for different platforms and thus may cause de facto inconsistencies which are likely to indicate JVM defects, e.g., the same class file may run smoothly on one JVM but trigger verifier errors on another JVM.

Testing JVMs via manually designing tests based on analyzing JVM semantics can be extremely challenging due to their intricacies, i.e., it is hard to generate sufficient high-quality inputs based on complicated JVM semantic rules to thoroughly test the program states of JVM executions. To address such a challenge, prior research work attempts to integrate fuzzing [47] and differential testing [43] for automated JVM testing, i.e., designing fuzzers to generate class files as tests for executing different JVMs such that their discrepant execution results (defined as *inter-JVM discrepancies* in this paper) can be used for testing analytics. For instance, *ClassFuzz* [39] fuzzes Java class files by mutating their modifiers or variable types to test the loading, linking, and initialization phases in JVMs. More recently, *Classming* [38] fuzzes live bytecode to mutate the control flows in class files to test deeper JVM execution phases (e.g., bytecode verifiers and execution engines) across multiple JVMs.

However, the power of the existing JVM fuzzers may not be fully leveraged since they fail to apply seed scheduling and mutator scheduling mechanisms which have become vital in enhancing fuzzing effectiveness. In particular, seed scheduling refers to aggressively selecting and mutating seeds to facilitate program bug/vulnerability exposure. Many coverage-guided fuzzers [7, 29, 53, 56, 61, 62] schedule seeds for mutation simply when executing them can increase code coverage. The existing JVM fuzzers, on the contrary, fail to leverage code coverage as seed scheduling guidance because they can hardly exploit the runtime coverage information for fuzzing since JVMs are likely to cause non-deterministic coverage at runtime due to their adopted mechanisms, e.g., parallel compilation and on-demand garbage collection [39]. Specifically, *ClassFuzz* only collects coverage information for initializing JVMs and *Classming* even exploits no coverage for fuzzing. Similarly, while scheduling mutators guided by code coverage has been proven effective recently [56, 69], the existing JVM fuzzers are restrained by selecting mutators uniformly under no guidance.

In this paper, we present *SJFuzz* (***S***cheduling for ***J***VM ***Fuzz***ing, in our *GitHub* repository [21]), a JVM fuzzing framework which applies seed and mutator scheduling mechanisms to facilitate the exposure of discrepant execution results among different JVMs, i.e., inter-JVM discrepancies, for JVM differential testing. Specifically, *SJFuzz* schedules seeding class files under two types of guidance—discrepancy and diversity. On one hand, *SJFuzz* retains the class files that can be executed to directly incur inter-JVM discrepancies or used to generate mutants for sufficiently testing JVMs, i.e., avoiding early termination on JVM testing process, as seeds for further mutations. On the other hand, assuming that increasing code coverage can be reflected by diversifying test case (class file) generation, *SJFuzz* applies a coevolutionary algorithm [9] to filter the remaining class files to augment class file diversity for further mutations. Moreover, *SJFuzz* also iteratively schedules mutators to augment the overall distances between seed and mutant class files. In particular, for a given seeding class file, *SJFuzz* estimates the diversity expectation of each mutator and selects a mutator to optimize the class file diversity.

To evaluate *SJFuzz*, we conduct a set of experiments upon various popular real-world JVMs, e.g., OpenJDK, OpenJ9, DragonWell, and OracleJDK. In particular, we apply *SJFuzz* and *Classming*, the state-of-the-art mutation-based JVM fuzzer, to generate class files via seeding class files selected from popular open-source Java projects, which are then executed in the studied JVMs to expose their discrepancies. Moreover, to further demonstrate the power of *SJFuzz*, we also include *JavaTailor* [82], a generation-based JVM fuzzer that utilizes existing JVM historical bug-revealing test programs to

```
1  protected Enumeration<URL> findResources(...){
2  +   i0 = 5
3      ...
4      r4 = r1.parent
5      ...
6  +   i0 = i0 + -1
7  +   if i0 <= 0 goto line 9
8  +   lookupswitch(i0) { case 4: goto line 4; default: goto line 13; }
9      r3 = $r5
10     ...
11     $z0 = r1.ignoreBase  // r1.ignoreBase is always 0
12     if $z0 == 0 goto line 19
13     $r7 = specialinvoke r1.getRootLoader()
14     if $r7 != null goto line 16
15     ...
16     $r8 = specialinvoke r1.getRootLoader()
17     $r9 = virtualinvoke $r8.getResources(r2)
18     ...
19     $r6 = staticinvoke CollectionUtils.append(r3, r14)
20     return $r6
21 }
```

**Figure 1: A class file generated under diversification guide.**

expose JVM discrepancies, as our baseline. The results suggest that *SJFuzz* significantly outperforms *Classming* in terms of inter-JVM discrepancy exposure, e.g., exposing 3.5×/6.3× more total/unique discrepancies on average. Meanwhile, *SJFuzz* also outperforms the *JavaTailor* by 5.2%/14.3% in terms of total/unique inter-JVM discrepancy exposure. Moreover, we have reported 46 potential issues to their corresponding developers after analyzing the inter-JVM discrepancies incurred by *SJFuzz*. As of submission time, 20 bugs have already been confirmed by the developers.

In summary, this paper makes the following main contributions:

- **Technique.** We introduce *SJFuzz*, which to the best of our knowledge is the first JVM fuzzing framework that applies seed and mutator scheduling mechanisms to test JVMs.
- **Implementation.** We implement our JVM testing approach as a practical system based on Jimple-level mutation via the Soot analysis framework [66].
- **Evaluation.** We conduct an extensive evaluation upon four popular JVMs and various real-world benchmark projects. The experimental results demonstrate that *SJFuzz* significantly outperforms the SOTA mutation-based and generation-based JVM fuzzers. Notably, we reported 46 potential issues found by *SJFuzz*, out of which 20 were confirmed and 16 were fixed by the developers.

## 2 MOTIVATING EXAMPLE

In this section, we introduce a real-world JVM bug exposed by applying differential testing via mutating program control flows to illustrate the potential issues of state-of-the-art *Classming* and motivate *SJFuzz*. Specifically, Figure 1 shows a simplified Jimple code snippet of a mutated method `findResources()` in `AntClassLoader.class` from project `Ant`, where the Jimple code representation refers to a Soot-based intermediate representation of Java programs for simplifying Java bytecode analysis [67]. Running such a class file exposes an execution discrepancy between OpenJDK (1.8.0_232) and OpenJ9 (1.8.0_232). Specifically, in the original seeding class file, after `z0` is assigned with the value of member `ignoreBase` of `r1`, i.e., 0 (line 11), line 12 is immediately executed, followed by line 19. However, inserting the `lookupSwitch` instruction changes the control flow to be from line 8 to line 13. Next, method `getResources()` of the root class loader is invoked (line 17). As

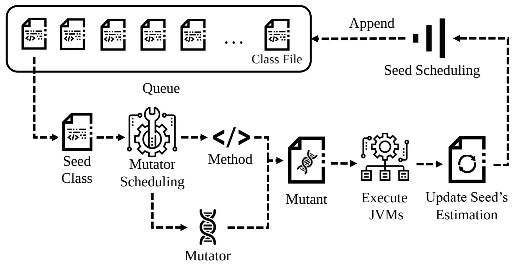



**Figure 2: The framework of *SJFuzz*.**

a result, by passing parameter `META-INF/MANIFEST.MF` to `getResources()`, OpenJDK8 (1.8.0_232) returns nothing while OpenJ9 (1.8.0_232) returns the paths of the `MANIFEST.MF` files in its lib JARs. We further found such a discrepancy was triggered as OpenJDK8 failed to find the existing resources.

Although this discrepancy can be exposed by simply inserting a `lookupSwitch` instruction to the seeding class file, *Classming* failed to expose such a discrepancy under multiple runs in practice. Specifically, we found that for the example class file, *Classming* updated new seeding class files after iteratively generating mutants via uniformly selected mutators, and failed to reproduce the discrepancy-inducing execution path under a fair time limit. This fact suggests that adopting similar seeding class files via only uniformly selected mutators may hinder the effective exploration of discrepancy-inducing mutants. Furthermore, we also observe that while *Classming* typically adopted similar seeding class files mutated from one root throughout causing inefficient usage of computing resources, which can be leveraged to explore other promising seeding class files, e.g., the ones that can expose multiple discrepancies [5, 6] simultaneously. This fact also leads to a demand of scheduling multiple seeding class files other than only one in one run based on their discrepancy-guided potentials.

To conclude, inspired by previous works [29, 34, 56, 77], all these insights motivate our proposed approach *SJFuzz*, which effectively schedules seeding class files and diversifies their mutations to increase the chances of exposing discrepancies for JVM testing.

## 3 THE APPROACH OF *SJFUZZ*

The framework of *SJFuzz* is demonstrated in Figure 2. Overall, *SJFuzz* enables iterative mutation-based class file generation. In particular, given a seeding class file, *SJFuzz* adopts the control flow mutation strategy to generate its mutant class file (Section 3.1). Accordingly, for each iteration, *SJFuzz* schedules seeding class files (Section 3.2) under the diversity and discrepancy guidance. *SJFuzz* also schedules mutators deterministically or randomly to augment class file diversity (Section 3.3) for further iterations.

Algorithm 1 shows the details of *SJFuzz*, which is initialized by adding one `seedClass` into the `queue` and assigning the `seedClass` to be *optional* (defined in Section 3.2). Under each iterative execution (line 6), *SJFuzz* schedules control-flow mutators for each class file in the `queue` to facilitate class file diversity (lines 7 to 8). Note that a newly generated class file is initialized as an *optional* seed (line 9). Such a seed and its parent (i.e., `mutantClass` and `class`) can be identified whether to be *primary* (defined in Section 3.2) after running on the adopted JVMs (lines 10 to 14). For any valid mutant class file, *SJFuzz* updates its distance to its seeding class file to guide

**Algorithm 1** The framework of *SJFuzz*

```
Input: seedClass, budget, bound
Output: queue
1:  function SJFUZZ_FRAMEWORK
2:      queue ← list()
3:      queue.add(seedClass)
4:      setClassToOptional(seedClass)
5:      while total budget has not exceeded do
6:          for class in queue do
7:              method ← randomlySelectMethod()
8:              mutantClass ← scheduleMutator(class, method)
9:              setClassToOptional(mutantClass)
10:             runJVMs(mutantClass)
11:             if mutantClass incurs new DISCREPANCIES then
12:                 setClassToPrimary(class)
13:                 if mutantClass is VALID then
14:                     setClassToPrimary(mutantClass)
15:             if mutantClass is VALID then
16:                 distance ← Levenshtein(class, mutantClass)
17:                 updateMutatorDistance(class, distance)
18:             else
19:                 updateMutatorDistance(class, -1)
20:         primary ← retainPrimarySeeds(queue, bound)  ▹ Discrepancy guidance
21:         option ← scheduleOptionalSeeds(queue, bound)  ▹ Diversity guidance
22:         queue ← merge(primary, option)  ▹ Retain seeds for next iteration
23:     return queue
```

further mutator scheduling (lines 16 to 17), and the distance for each invalid mutant class file is updated to -1 (line 19). At last, all the *primary* class files and filtered *optional* class files are retained for future mutations (as the output of the discrepancy guidance and the diversity guidance, lines 20 to 22). After each iteration, the updated seeding class files are used for JVM differential testing. Such iterations are terminated when hitting the `budget`. Note that *SJFuzz* only enables valid class files for mutations because mutating an invalid class file tends to cause exceptional program behaviors rather than unexplored inter-JVM discrepancies.

### 3.1 Control Flow Mutation

Prior research work on fuzzing compilers including JVMs tend to mutate program control flows via a set of corresponding mutators for exposing bugs in their "deep" execution stages [31, 38, 41, 50]. Following such prior works (and also for a fair comparison with them), *SJFuzz* also adopts such control flow mutation with representative mutators. Specifically in source code level, *SJFuzz* randomly selects two original instructions, and creates a directed transition between them. If such a transition is a loop, the corresponding iteration will be limited to 5 times. Correspondingly, *SJFuzz* implements the mutators with the Jimple-level instructions `goto`, `lookupswitch`, and `return` provided in *Soot* [66].

*SJFuzz* iteratively selects random positions from randomly selected methods to apply the control-flow mutators. Specifically, *SJFuzz* establishes an instruction list which contains the instructions executed by the adopted JVMs under their execution order. Next, *SJFuzz* selects and inserts a control flow mutator into a random spot of the instruction list under each iteration.

## 3.2 Seed Scheduling

Since it is difficult to directly apply coverage guidance for fuzzing JVMs, we adopt two alternative types of guidance for our seed scheduler. In particular, we develop a discrepancy-guided seed scheduler which retains discrepancy-inducing class files for further mutations. We also develop a diversity-guided seed scheduler which filters other class files to augment the overall class file diversity via a coevolutionary algorithm [9].

**Discrepancy-guided seed scheduling.** Intuitively, if running a mutant of a class file can cause inter-JVM discrepancies, such a class file is likely to generate more discrepancy-inducing mutants than other class files as it implies a potential connection with program bugs [5, 6]. Therefore, such a class file and its mutant (if valid, i.e., successfully running in at least one JVM under test without unexpected behaviors such as verifier errors or crashes) are defined as *primary* class files and are retained for future iterative executions. This seed scheduler is essentially similar to many coverage-guided seed schedulers [7, 29, 53, 61, 62], which tend to retain seeds when running them can increase code coverage.

**Diversity-guided seed scheduling.** When running a mutant of a class file does not instantly cause inter-JVM discrepancies, it does not necessarily suggest that no discrepancy-inducing mutants can be generated in future iterative executions. In other words, leveraging such class files can also possibly advance the inter-JVM discrepancy exposure. In this paper, such class files are characterized as *optional*. Note that differential testing usually enables vast space for generating test cases, which indicates that the total number of class files that cause discrepancies between sophisticated JVMs can be rather limited. We can then infer that the *optional* class files may significantly outnumber the primary class files. Therefore, *SJFuzz* should filter the *optional* class files to ensure fuzzing efficiency.

We consider that increasing code coverage can be essentially reflected as diversifying execution paths on the same JVMs and thus demands diverse test cases (i.e., class files). Therefore, our seed scheduler for *optional* class files is guided by class file diversity, so that the *optional* class files are filtered to augment class file diversity. In particular, how to measure class file diversity should be resolved in the first place. To accurately reflect the fine-grained differences between class files, an ideal metric is expected to reflect their instruction-by-instruction comparisons. Therefore, we adopt the executed instruction list of a given class file, namely *EntryInstruction*, as the representative instructions to efficiently measure the diversity between JVM class files. Note that *EntryInstruction* reflects the instruction-level execution order and retains only the unique executed instructions to reduce the ambiguity of diversity measurement caused by repeated instructions, e.g., in loops. Eventually, we measure the diversity between a pair of class files by deriving differences between their associated *EntryInstruction*s.

In this paper, *SJFuzz* applies edit distance, i.e., Levenshtein Distance [52], a metric widely used to derive the "minimum number of single-character edits (insertions, deletions, or substitutions)" between two strings, to measure the difference between *EntryInstructions* because the mutation-based class file generation can analogize the single-character string edits as demonstrated in [35]. To be specific, *SJFuzz* generates class files by mutating the selected seeding class files, i.e., inserting the instructions with the adopted control flow mutators. The resulting iterative single-point mutations between the seed and mutant class files can be modeled as inputs for Levenshtein-Distance-based computation when such class files are all modeled as "strings". For instance, assume two *EntryInstructions* of their corresponding class files $C_1 : [i_1, i_2, i_3, i_4, ..., i_n]$ and $C_2 : [i_1, i_3, i_4, ..., i_n]$. We can observe that $C_1$ can be transformed from $C_2$ by only inserting one instruction $i_2$ between $i_1$ and $i_3$. Therefore, their Levenshtein Distance is computed as 1.

Accordingly, *SJFuzz* adopts a coevolutionary algorithm [9] to efficiently evaluate the individual *optional* class files out of their group by constructing its fitness function to reflect their average distances with other *optional* class files. Specifically, for each *optional* class file, *SJFuzz* calculates its total Levenshtein Distance with other *optional* class files. Subsequently, the average Levenshtein Distance is calculated as the fitness score of the given class file. By sorting all the derived fitness scores, *SJFuzz* retains the top-N corresponding *optional* class files for further mutation-based class file generation, where N is predefined as the bound variable in Algorithm 1.

To illustrate, we incorporate the discrepancy- and diversity-guided seed schedulers to facilitate code coverage and inter-JVM discrepancies when differentially testing JVMs.

## 3.3 Mutator Scheduling

Since it is computationally expensive to derive the exact diversity of the overall class files on the fly, *SJFuzz* schedules mutators to diversify the seed and mutant class files under each iteration to approximate the overall class file diversity instead. In particular, *SJFuzz* first applies the edit distance, i.e., Levenshtein Distance [52] in this paper, to delineate the diversity between a pair of class files. Accordingly, *SJFuzz* establishes a *deterministic mutator scheduling mechanism* for estimating the mutator that can optimize the seed-mutant distance. Meanwhile, *SJFuzz* also develops a *random mutator scheduling mechanism* to prevent the potential local optimization that can derive local optimal mutators caused by the *deterministic mutator scheduling mechanism*. As a result, *SJFuzz* derives a mutator for a given class/method by combining the two mechanisms.

**Deterministic mutator scheduling.** Note that any mutator selected from one iteration can incur cumulative impact on the mutations of the subsequent iterations. To capture such cumulative impact from the previous mutations, *SJFuzz* adopts the *Monte Carlo method* [22] to develop the *deterministic mutator scheduling mechanism*, where given a selected method $m_p$, *SJFuzz* develops a value function, represented as $V(c_i, a_j, m_p)$, to determine the mutation opportunity of class file $c_i$ by applying a mutator $a_j$ as demonstrated in Equation 1. Such a value function can reflect the resulting diversity of the overall class files under the mutation, i.e., the cumulative diversity expectation between the seed and mutant class files under all the iterations.

$$V(c_i, a_j, m_p) = \mathbb{E}[\frac{1}{N} \sum_{k=0}^{N} Distance_k(c_i, a_j, m_p)] \quad (1)$$

Here $Distance$ refers to the Levenshtein Distance between the seeding class file $c_i$ and its mutant class file by applying mutator $a_j$ upon method $m_p$. $Distance_k(c_i, a_j, m_p)$ refers to their Levenshtein Distance in the $k_{th}$ iteration which can be dynamically updated

**Algorithm 2** Mutator Scheduling

**Input** : class, method, explorationRate
**Output**: mutantClass

1: **function** SCHEDULE_MUTATOR
2: rand ← *Random*()
3: **if** rand < explorationRate **then**
4: mutator ← *selectRandomMutator*(class, method)
5: mutatedClass ← *mutate*(mutator, class) ▷ Generate a mutant by a mutator
6: **else**
7: bestMutator ← *selectDeterministicMutator*(method)
8: mutatedClass ← *mutate*(bestMutator, class)
9: **return** mutatedClass

since the mutation spot is randomly selected in $m_p$ under each iteration. $\mathbb{E}$ refers to the mathematical expectation of $Distance$s. It can be derived that $V(c_i, a_j, m_p)$ for $c_i$ is incrementally updated and inefficient to be directly computed. Therefore, we further enable dynamic updates on $V_k(c_i, a_j, m_p)$ as presented in Equation 2 to approximate its value, where $\alpha$ is a constant. Note when one class file $c_i$ fails to generate a valid class, its $Distance$ is set to $-1$.

$$V_k(c_i, a_j, m_p) = V_{k-1}(c_i, a_j, m_p) + \alpha(Distance - V_{k-1}(c_i, a_j, m_p)) \tag{2}$$

As a result, we select a mutator $a_j$ corresponding to the largest $V(c_i, a_j, m_p)$ for $c_i$. Typically, *SJFuzz* allows computing $V(c_i, a_j, m_p)$ after running the mutant class files on JVMs such that it can be used for mutator selection of the subsequent iteration when needed (as in line 17 of Algorithm 1).

**Random mutator scheduling.** Only maximizing $V(c_i, a_j, m_p)$ tends to cause local optimization, i.e., $V(c_i, a_j, m_p)$ is likely to converge to one mutator after iteratively selecting it, while the actual optimal mutator cannot be derived until later iterations. To address such an issue, *SJFuzz* further leverages a *random mutator scheduling mechanism* to reduce its possibility to select a sub-optimal mutator under early-terminated executions by randomly selecting one mutator for class file generation for the ongoing iteration. As a result, by properly combining such a mechanism with the *deterministic mutator scheduling mechanism*, it can potentially extend the *Monte Carlo* process until convergence for enhancing the selection probability of the optimal mutator, i.e., preventing the local optimization.

The overall mutator scheduling mechanism is presented in Algorithm 2. We first set an `explorationRate` and generate a random value for comparison (line 2). Next, if such a random value is less than the `explorationRate`, *SJFuzz* chooses the *random mutator scheduling mechanism* to return a random mutator for the ongoing iteration (lines 3 to 5). Otherwise, *SJFuzz* derives the mutator via the *deterministic mutator scheduling mechanism* (lines 6 to 8).

# 4 EVALUATION

We conduct a set of experiments on various popular JVMs. Note that we include two state-of-the-art JVM fuzzers—the mutation-based fuzzer *Classming* [38] and the generation-based fuzzer *JavaTailor* [82] in our evaluation for performance comparison. In particular, *Classming* fuzzes live bytecode to mutate the control flows in class files as mentioned, and *JavaTailor* generates class files from JVM historical bug-revealing test programs (provided by the authors). Overall, we aim to compare *SJFuzz* with them in terms of their resulting inter-JVM discrepancies, the class file generation efficiency, and the reported bugs by answering the following research questions:

- **RQ1:** Is *SJFuzz* effective in exposing inter-JVM discrepancies?
- **RQ2:** Are the seed and mutator schedulers effective?
- **RQ3:** Is the diversity guidance effective?

Moreover, we report and analyze the bugs detected by *SJFuzz* with all the evaluation details presented in our GitHub page [21].

## 4.1 Benchmark Construction

We adopt multiple widely-used real-world JVMs, i.e., OpenJDK, OpenJ9, DragonWell, and OracleJDK, for running *SJFuzz* to expose their execution discrepancies. Note that their detailed versions are available on our GitHub page [21] since multiple versions of each JVM are used in our evaluation. We also adopt state-of-the-art mutation-based approach *Classming* and generation-based approach *JavaTailor* as the baselines for comparison as they outperformed other existing JVM differential testing approaches [38, 82]. Specifically, for a fair comparison with *SJFuzz* which integrates differential testing and test generation, we also run *Classming* and *JavaTailor* on all studied JVMs in parallel.

To launch *SJFuzz*, we adopt 26 class files via randomly sampling from 7 well-established open-source projects as the seeding class files for mutation-based class file generation. To construct such benchmarks, we first attempt to collect all available class files originally adopted for evaluating *Classming*, for approaching a fair performance comparison. As a result, `Eclipse`, `Jython`, `Fop`, and `Sunflow` are selected due to their availability while others incur stale configurations, JAR incompatibility, mismatched `main` declarations, etc. Moreover, we also adopt `Ant` and `Ivy` (two popular command-line applications from Apache Projects [8]) and `JUnit` [19] (a widely used unit testing framework) to expand our benchmark diversity.

Note that while the existing approaches, e.g., *Classming* and *ClassFuzz*, are designed to only launch mutations for the entry methods corresponding to the `main` methods, in this paper, we attempt to adopt diverse "entry" modes, i.e., diverse method types (entries) for mutation. Particularly, we adopt two such modes: *main-entry* and *JUnit-entry*. More specifically, in addition to *main-entry* adopted by [38, 39], the new *JUnit-entry* mode, on the other hand, refers to mutating other entry methods associated with JUnit tests of the seeding class files. To our best knowledge, we are the first to execute the unit tests of the seeds in compiler testing.

*JUnit-entry* can benefit the class file generation for the following reasons. First, *JUnit-entry* supplements *main-entry* on the mutation space for a class file which cannot be explored by *main-entry* only, since a large amount of JUnit test classes are designed for non-main methods in practice. Next, the execution discrepancies between JVMs are likely to be better presented in *JUnit-entry*, since assertions examine different JVM executions upon class files and thus enable smaller scope in exposing discrepancies and easier analytics than *main-entry*.

In this paper, for *main-entry*, we select seeding class files as the class files containing the `main` methods. For *JUnit-entry*, since each project contains various test classes, we randomly adopt 4 class files under test for each project with more than 5 corresponding test methods from the projects `Fop`, `Jython`, `Ant`, `Ivy`, and `JUnit` which all use the JUnit framework with available test source files on the corresponding GitHub repositories. Note that `Fop`, `Jython`, `Ant`, and `Ivy` are chosen as both the *main-entry* and *JUnit-entry* benchmarks for straightforward performance comparison between the two modes within one project.

## 4.2 Environmental Setups

We perform our evaluation on a desktop machine, with Intel(R) Xeon(R) CPU E5-4610 and 320 GB memory. The operating system is Ubuntu 16.04. The `exploreRate` for Algorithm 2 is set to 0.1 and the `bound` for Algorithm 1 is set to 20 by default.

Similar as prior work [28, 29, 39, 61, 62], all benchmarks are executed by all the studied approaches for 24 hours to generate class files to reflect a large enough testing `budget`. Note that we run each experiment 20 times [49] for obtaining the average results to reduce the impact of randomness.

## 4.3 Result Analysis

**Table 1: Discrepancies exposed by *SJFuzz*, *Classming* and *JavaTailor*.**

| Project | Benchmark(.class) | *SJFuzz* | | *Classming* | | *JavaTailor* | |
|---|---|---|---|---|---|---|---|
| | | Total | Unique | Total | Unique | Total | Unique |
| Eclipse | EclipseStarter | 2776.4 | 9.8 | 248.2 | 1.0 | 2487.2 | 8.3 |
| Fop | Fop | 22.3 | 1.9 | 1.0 | 1.0 | 21.1 | 1.4 |
| Jython | Jython | 800.5 | 7.0 | 14.1 | 1.0 | 874.3 | 7.4 |
| Sunflow | Benchmark | 1764.9 | 8.7 | 348.8 | 1.0 | 1574.2 | 6.0 |
| Ant | Launcher | 2514.6 | 11.8 | 486.1 | 1.0 | 2587.9 | 12.8 |
| Ivy | Main | 5042.9 | 19.1 | 1557.9 | 2.8 | 4679.9 | 14.9 |
| Fop (JUnit) | FopConfParser | 1612.1 | 9.9 | 84.3 | 0.8 | 1417.2 | 7.8 |
| | FopFactoryBuilder | 1651.2 | 9.3 | 382.9 | 3.0 | 1844.2 | 9.7 |
| | ResourceResolverFactory | 214.2 | 3.0 | 22.2 | 1.0 | 203.9 | 2.5 |
| | FontFileReader | 796.3 | 5.8 | 111.9 | 1.0 | 791.4 | 5.3 |
| Jython (JUnit) | PyByteArray | 2142.7 | 10.1 | 932.4 | 1.9 | 2209.2 | 10.3 |
| | PyFloat | 928.7 | 4.2 | 260.9 | 1.0 | 897.1 | 3.1 |
| | PySystemState | 232.4 | 3.8 | 56.5 | 1.0 | 222.3 | 3.0 |
| | PyTuple | 1661.8 | 7.2 | 312.6 | 1.1 | 1434.9 | 5.0 |
| Ant (JUnit) | AntClassLoader | 298.1 | 10.7 | 6.2 | 1.0 | 290.9 | 9.8 |
| | DirectoryScanner | 71.5 | 8.9 | 1.8 | 1.0 | 67.0 | 7.8 |
| | Project | 53.1 | 1.8 | 0.0 | 0.0 | 49.9 | 1.4 |
| | Locator | 1573.7 | 8.9 | 447.1 | 1.0 | 1485.4 | 7.8 |
| Ivy (JUnit) | ResolveReport | 211.7 | 4.7 | 0.0 | 0.0 | 197.7 | 4.0 |
| | ApacheURLLister | 645.9 | 7.2 | 213.4 | 1.0 | 605.2 | 6.2 |
| | Configurator | 509.0 | 7.1 | 57.5 | 1.0 | 470.9 | 6.0 |
| | IvyEventFilter | 1218.6 | 8.6 | 386.2 | 2.0 | 1109.7 | 7.7 |
| JUnit (JUnit) | RuleChain | 972.3 | 14.3 | 264.9 | 1.0 | 763.3 | 9.5 |
| | TestWatcher | 817.9 | 2.2 | 228.6 | 1.0 | 786.6 | 1.6 |
| | ErrorReportingRunner | 1986.6 | 10.6 | 506.9 | 1.0 | 1872.2 | 9.3 |
| | Money | 914.3 | 11.2 | 88.3 | 1.0 | 914.4 | 13.0 |
| Average | | 1208.0 | 8.0 | 270.0 | 1.1 | 1148.4 | 7.0 |
| p-value | | N/A | | 2.35e-84 | | 0.039 | |

### 4.3.1 *RQ1: the Inter-JVM Discrepancy Exposure Effectiveness of SJFuzz.*

Note that in this paper, to identify unique discrepancies, we first summarize the symptoms of the discrepant JVM behaviors, including assertions in JUnit tests, exceptions, and the results printed in standard output. Then we compare such symptoms with the previously recorded unique discrepancies to distinguish whether they are unique or not. More specifically, we first divide the overall output results into two categories—*non-exception output* (e.g., assertions in JUnit tests and results printed in standard output) and *exception*. Typically, an exception can be respectively presented for different JVMs. We then represent an exception as a tuple of its type and location from the output result. Furthermore, given one exception on all tested JVMs, one discrepancy is formed by collecting, analyzing, and combining all their exception information. If no such a discrepancy was collected before, it is considered unique. The implementation code for this process can be found at [10].

The inter-JVM discrepancy results (both the total and the unique discrepancies) after executing the generated class files are presented in Table 1. For instance, for benchmark `Jython.class`, *SJFuzz* exposed a total of 800.5 discrepancies and 7.0 unique discrepancies averagely. We can observe that overall, *SJFuzz* can significantly outperform *Classming* in terms of the inter-JVM discrepancy exposure. To be specific, *SJFuzz* can expose 1208.0 total discrepancies and 8.0 unique discrepancies on average, while *Classming* can expose 270.0 total discrepancies and 1.1 unique discrepancies on average, i.e., *SJFuzz* exposes over 3.5×/6.3× more total/unique discrepancies than *Classming*. Moreover, we can further find that for all the adopted benchmark projects, *SJFuzz* can significantly outperform *Classming* in terms of both the total and unique discrepancy exposure. Note that *SJFuzz* also exposes all the discrepancies found by *Classming* in our evaluation. Meanwhile, we can observe that *SJFuzz* can also outperform *JavaTailor* by 5.2% more total discrepancies (1208.0 vs. 1148.4) and 14.3% more unique discrepancies (8.0 vs. 7.0) respectively.

Furthermore, we apply the Mann-Whitney U test [57] to illustrate the significance of *SJFuzz*. It can be seen in Table 1 that the $p$-value of *SJFuzz* comparing with *Classming* in terms of the average unique discrepancies is far below 0.05 in each benchmark, which indicates that *SJFuzz* outperforms *Classming* significantly ($p < 0.05$). We can also observe that the $p$-value of *SJFuzz* comparing with *JavaTailor* in terms of the average unique discrepancies is also below 0.05 (0.039). Such results can reflect that *SJFuzz* can be quite effective.

Interestingly, we can observe that the advantage of *SJFuzz* over *JavaTailor* is not quite obvious as over *Classming*, i.e., *JavaTailor* is a more powerful baseline. We infer it is mainly because *JavaTailor* adopts a database containing a variety of historical JVM-bug-revealing test programs which can be quite enlightening for testing tasks. Nevertheless, as a typical data-driven approach, it can be naturally prone to common issues, e.g., data dependency and extra effort on maintaining the database. Surprisingly, *SJFuzz*, a lightweight end-to-end approach, can still outperform *JavaTailor* in exposing both total and unique discrepancies, indicating the power of our adopted mechanism of seed and mutator scheduling.

> **Finding 1:** *SJFuzz is effective by exposing 6.3× more unique discrepancies averagely than Classming (8.0 vs. 1.1), and 14.3% more unique discrepancies averagely than JavaTailor (8.0 vs. 7.0) under the same evaluation setups.*

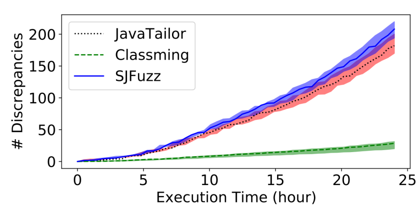
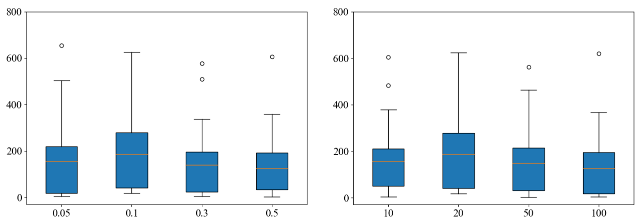



**Figure 3: *SJFuzz*/*Classming*/*JavaTailor* efficiency in 24 hours.**

We further investigate the impact of the execution time on discrepancy exposure by *SJFuzz*, *Classming*, and *JavaTailor*. Figure




**(a) Exploration Rate.** **(b) Bound.**

**Figure 4: The impact of the parameter settings on *SJFuzz* in all benchmarks.**

3 shows how the exposed unique discrepancies on all the benchmarks by the three approaches vary over time. We can observe that although we enhanced the differential testing efficiency by running JVMs in parallel for *Classming* and *JavaTailor* (as in Section 4.1), we can also observe that *SJFuzz* can in general consistently outperforms *Classming* and *JavaTailor* in finding JVM discrepancies all the time before terminating the executions. Such results can further indicate the power of the discrepancy guidance mechanism of *SJFuzz*.

We also investigate the discrepancies exposed by our adopted entry modes for class file mutations: *main-entry* and *JUnit-entry*. Specifically, *SJFuzz* can significantly outperform *Classming* under both the entry modes, i.e., *SJFuzz* can expose 58.3 unique discrepancies in 6 *main-entry* benchmarks and 149.5 unique discrepancies in 20 *JUnit-entry* benchmarks, while *Classming* can only expose 7.8 and 21.6 unique discrepancies under such two entry modes respectively. *JavaTailor* also exposes 50.8 unique discrepancies in 6 *main-entry* benchmarks and 130.8 unique discrepancies in 20 *JUnit-entry* benchmarks. Additionally, for the projects which enable both *main-entry* and *JUnit-entry* (i.e., `Fop`, `Jython`, `Ant`, and `Ivy`), *SJFuzz* exposes 111.2 unique discrepancies in total under *JUnit-entry* and 39.8 under *main-entry*. Such a result indicates the effectiveness of our newly proposed *JUnit-entry* mode for JVM testing. We highly encourage future researchers/practitioners to look into the *JUnit-entry* mode for advancing JVM testing.

> **Finding 2:** *JUnit-entry is more effective than main-entry in exposing inter-JVM discrepancies.*

We have also observed that *SJFuzz* has rather stable performance across different configurations. To evaluate the impact of the parameter settings on *SJFuzz*, we evaluate the unique discrepancies in terms of different `explorationRate` (in Algorithm 2) and `bound` (in Algorithm 1) values on our benchmark suite, as presented in Figure 4. In particular, we set `bound` to the default 20 and investigate the impact of different `explorationRate`, i.e., 0.05, 0.1, 0.3 and 0.5 (Figure 4a). We also set `explorationRate` to the default 0.1 and investigate the impact of different `bound` (Figure 4b). Each box plot presents the distribution of exposed unique discrepancies for one configuration across all our studied subjects. We can observe that different configurations exert limited impact on the performance, indicating the effectiveness and stability of *SJFuzz*.

*4.3.2 RQ2: Effectiveness of the Seed and Mutator Schedulers.* In this section, we investigate the effectiveness of the seed and mutator schedulers respectively.

**Effectiveness of the seed scheduler.** To investigate the effectiveness of the adopted seed scheduler of *SJFuzz*, we record the number of discrepancies exposed by the *primary* and *optional* class files of the original *SJFuzz* approach, denoted as *SJFuzz*(primary) and *SJFuzz*(optional), respectively. Furthermore, we also build the two variant techniques of *SJFuzz*: (1) $SJFuzz_{pg}$, which only activates discrepancy-guided seed scheduling for *SJFuzz*, and (2) $SJFuzz_{ef}$, which equally filters the class files regardless whether they are *primary* or *optional*. Note that $SJFuzz_{pg}$ retains the initial class file for further mutations until it explores a *primary* class file given that the initial class file is not *primary*.

In general, we can observe from Table 2 that $SJFuzz_{pg}$ can be effective by exposing 252.3 total discrepancies and 1.2 unique discrepancies on average. Interestingly, only $SJFuzz_{pg}$ itself can enable quite close performance with *Classming* (270.0 total discrepancies and 1.1 unique discrepancies on average as in Table 1). Such results can indicate the effectiveness of our "discrepancy-guided" intuition, i.e., exploiting the power of discrepancy-inducing class files can advance JVM differential testing.

Interestingly, Table 2 demonstrates that by integrating $SJFuzz_{pg}$ and $SJFuzz_{ef}$, i.e., applying the original *SJFuzz*, mutating *primary* class files can incur significantly more inter-JVM discrepancies, i.e., 381.3 vs. 252.3 total discrepancies with 2.9 vs. 1.2 unique discrepancies between *SJFuzz*(primary) and $SJFuzz_{pg}$. Such results indicate that injecting *optional* class files for test case generation can advance the *primary* class files to generate more discrepancy-inducing class files. To illustrate, when mutating *optional* class files generate discrepancy-inducing mutant class files, they are all converted to be *primary*. Thus, *primary* class files are increasingly adopted for further mutations such that their chances to expose discrepancies can be augmented. Furthermore, the fact that *SJFuzz*(optional) outperforms $SJFuzz_{ef}$ suggests that directly retaining *primary* class files for further mutations can also advance the *optional* class files to expose inter-JVM discrepancies. We can infer that by independently mutating *primary* class files via revoking their filtering process, more *optional* class files can be retained for further mutations because the *primary* class files no longer compete against them for being selected. To summarize, $SJFuzz_{pg}$ and $SJFuzz_{ef}$ can mutually advance each other to optimize the performance of *SJFuzz*.

> **Finding 3:** *As different components of the seed scheduling mechanism, $SJFuzz_{pg}$ and $SJFuzz_{ef}$ are both effective and integrating them can further advance each other in terms of exposing inter-JVM discrepancies.*

**Effectiveness of the mutator scheduler.** To investigate the effectiveness of the mutator scheduler of *SJFuzz*, we build a variant technique $SJFuzz_{uniform}$ of the original *SJFuzz* by selecting mutators uniformly. Overall, we can observe from Table 2 that $SJFuzz_{uniform}$ can expose 819.6 total discrepancies and 5.4 unique discrepancies averagely. Although $SJFuzz_{uniform}$ still outperforms *Classming* 2.0×/3.9× averagely in exposing total/unique discrepancies, the exposed discrepancies decrease significantly after disabling

**Table 2: Average number of discrepancies found by the studied techniques upon all benchmark projects.**

| Studied Subjects | All Discrepancies | Unique Discrepancies |
|---|---|---|
| *SJFuzz*(primary) | 381.3 | 2.9 |
| *SJFuzz*(optional) | 826.7 | 5.1 |
| $SJFuzz_{pg}$ | 252.3 | 1.2 |
| $SJFuzz_{ef}$ | 764.7 | 4.8 |
| *JavaTailor* | 1148.4 | 7.0 |
| $SJFuzz_{uniform}$ | 819.6 | 5.4 |
| *Classming* | 270.0 | 1.1 |
| *SJFuzz* | 1208.0 | 8.0 |

mutator scheduling, i.e., 32.2%/32.5% averagely in exposing total/unique discrepancies compared to *SJFuzz*. Moreover, *JavaTailor* can outperform this variant by 40.1%/29.6% averagely in exposing total/unique discrepancies. Such results indicate that applying the mutator scheduler in *SJFuzz* can make significant contributions in terms of inter-JVM discrepancies.

> **Finding 4:** *Applying the mutator scheduling mechanism can significantly improve the power of exposing discrepancies for JVM fuzzers.*

*4.3.3 RQ3: Effectiveness of the Diversity Guidance.* The previous findings of the effectiveness of different *SJFuzz* components can imply their underlying mechanism of diversifying class file generation can be potentially effective. However, accurately measuring data diversity can be rather challenging. In this paper, we delineate class file diversity in terms of the average seed-mutant Levenshtein Distance of the collected class files. Note that since *JavaTailor* is a generation-based approach (i.e., generating new class files with a variety of JVM-bug-revealing test programs can naturally result in significantly large Levenshtein Distances), we thus only include *Classming* in this discussion.

The diversity results of the class file generation are presented in Figure 5. We can observe that *SJFuzz* incurs much larger average seed-mutant Levenshtein Distance compared with *Classming*, i.e., overall 23.7× larger and 32.3×/22.5× larger under *main-entry*/*JUnit-entry*. It can be inferred that *Classming* tends to generate similar mutants, which also indicates the effectiveness of *SJFuzz*'s diversity-guided class file generation mechanism.

We further attempt to infer the possible reasons behind the diversity performance difference between *SJFuzz* and *Classming* in terms of the seed-mutant Levenshtein Distance. Assume a mutated class file (simplified version) in Figure 6 with only one executed instruction (line 4). Initially, *Classming* would select and insert the `return` mutator (line 5) because other mutators can result in the potential def-use violation of the `r0`-exclusive variables and thus the verification error. Such an error can hinder the detection of "deep" bugs, e.g., bugs incurred in execution engine. However, under this circumstance, the class file in Figure 6 is likely to be retained as the seed to repeatedly select the `return` mutator under each iterative execution for further class file generation. As a result, all the mutant class files realize single-mutation difference with their respective seeds, i.e., leading to potential short seed-mutant Levenshtein Distance. In contrast, *SJFuzz* is free from such constraints because it can diversify seed *optional* class files with best effort.

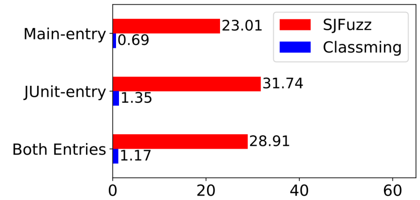



**Figure 5: Average seed-mutant Levenshtein Distance.**

```
1 class A {
2     ...
3     public void someFunction() {
4         r0=<java.lang.System: java.io.PrintStream out>;
5         return; // inserted by return mutator
6         ......
7     }
8 }
```

**Figure 6: An example of mutating paradox for *Classming*.**

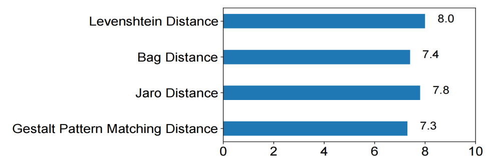



**Figure 7: Average unique discrepancies exposed by different distance metrics in all benchmarks.**

Moreover, even when a seed *optional* class file generates a similar mutant class file, they together are hardly retained for further mutations under the diversity-guided class file filtering mechanism.

> **Finding 5:** *Diversifying class file generation is advanced in testing the "deep" bugs in execution engine by retaining sufficient valid class files.*

At last, we study the effectiveness of our adopted distance metric, i.e., the Levenshtein Distance, which is used to measure class file diversity. To this end, we adopt more distance metrics [13, 17, 27] for discussing their impact on *SJFuzz*. Specifically, we adopted Gestalt Pattern Matching Distance [13], Bag Distance [27], and Jaro Distance [17] in our evaluation. In particular, Gestalt Pattern Matching Distance adopts the number of matching characters and the length of strings to measure the similarity for two given strings. Bag Distance utilizes the maximum length between the relative complements [24] of two given strings' character sets with respect to each other to measure their distance. Jaro Distance is also a type of edit distance to calculate the similarity of two strings. As shown in Figure 7, distance metrics exert limited impact on the effectiveness of *SJFuzz* in terms of exposing unique discrepancies. Interestingly, we can observe that adopting edit distances, i.e., Jaro Distance and Levenshtein Distance, can achieve slightly better performance than other distance metrics.

## 4.4 Bug Report and Discussion

We manually analyze all the collected discrepancies to derive potential issues. Note that in this paper, we define a bug as an error or an unexpected behavior for a specific JVM version. As a result, we report 46 potential issues from the discrepancies found by *SJFuzz* (*Classming* fails to expose any of these issues while *JavaTailor* can

**Table 3: Issues found by *SJFuzz*.**

| JVMs | # Reported | | | | # Confirmed | | | |
|---|---|---|---|---|---|---|---|---|
| | Loading Phase | Linking Phase | Run-time | Crash | Loading Phase | Linking Phase | Run-time | Crash |
| OracleJDK | 0 | 0 | 3 | 0 | 0 | 0 | 2 | 0 |
| OpenJDK | 2 | 3 | 3 | 3 | 0 | 0 | 2 | 0 |
| Dragonwell | 1 | 2 | 2 | 0 | 0 | 0 | 0 | 0 |
| OpenJ9 | 6 | 7 | 12 | 2 | 0 | 4 | 10 | 2 |
| **TOTAL** | 9 | 12 | 20 | 5 | 0 | 4 | 14 | 2 |

expose 25 of them), as in Table 3, to their corresponding developers. As of today, 20 were confirmed while 16 were fixed by the developers. The remaining 4 confirmed bugs are marked as "won't fix". We present some example bug reports as follows.

*4.4.1 Resource Retrieval Bug.* We reported an OpenJDK bug on retrieving JAR information which was confirmed by the OpenJDK developers with a bug ID `JDK-8244083`. This bug was exposed by the execution discrepancy between OpenJ9 and OpenJDK. Specifically, they both executed one class file from `AntClassLoader.class`, where OpenJDK failed to retrieve the JAR information from given resources while OpenJ9 succeeded. The developers inferred that certain side effect changed the behaviors of the original method.

*4.4.2 Runtime Inconsistency Bug.* We have reported an OpenJ9 bug on issuing a runtime erroneous return under the mutated classes from `Money.class`, as shown in Figure 8. We applied its original JUnit tests on all the class files mutated from `Money.class` which resulted in multiple errors/discrepancies. In particular, OpenJ9 reported an `AssertionError` while OpenJDK passed the test. However, when we further removed one JUnit test which caused `StackOverflowError`, both OpenJ9 and OpenJDK passed the test. Accordingly, we summarized that such a discrepancy may be caused by the unresolved dependency between JUnit tests and reported it to the corresponding developers [1].

To tackle such an issue, developers applied option `optlevel` at the `warm` level and inferred this as a JIT issue. After checking the `tree simplification` (an optimization feature in OpenJ9), developers found that OpenJ9 made a wrong assumption to the `nodeIsNonZero` flag set. As a result, the instruction `ificmpne` was changed to `goto` by OpenJ9 and it caused the associated branch to be always executed, even when the value of the associated variable did not meet the branch conditions. Eventually, they fixed this issue as follows:

> *...the* `nodeIsNonZero` *flag was set because IL gen assumed that slot 0 was still being used to store the receiver and thus the flag did not need to be reset. There is a method that is supposed to check if slot 0 was re-used so that flags can be reset. This problem can be fixed by adding cases to handle other types of stores to slot 0... I will open a pull request to make this change.*

*4.4.3 Verifier Bug.* A verifier bug usually is derived by analyzing the discrepancies about throwing a `verifyError` or not. In particular, verifier bugs are perceived typical "deep" bugs, i.e., bugs that are tricky to be detected and debugged.

By executing the mutated `ErrorReportingRunner.class` from project `JUnit`, we discovered that OpenJDK (1.8.0_232), OpenJDK (9.0.4), and OpenJDK (11.0.5) threw `VerifyError`, while OpenJ9 (1.8.0_232) and OpenJ9 (11.0.5) wrongly took it as a valid class file

```
class A {
    ...
    public boolean isZero() {
        int var1 = this.amount();
        // OpenJ9 and OpenJDK get var1 = 0 here
        boolean var2;
        if (var1 == 0) {
            var2 = true; // OpenJDK executed here
        } else {
            var2 = false; // OpenJ9 executed here
        }
        return var2;
    }
}
```

**Figure 8: Runtime inconsistency bug in OpenJ9.**

for execution. Moreover, there even incurred a discrepancy among multiple OpenJ9 versions, i.e., OpenJ9 (9.0.4) threw a `VerifyError`. Accordingly, we inferred that OpenJ9 (1.8.0_232) and OpenJ9 (11.0.5) were buggy and reported them to developers.

Interestingly, it took the developers quite a while to understand the cause of such bugs. At first, they speculated this issue as an "out of sync" problem:

> *It seems the code in verifier is likely out of sync or some new changes related to verifier were only merged for OpenJDK8 & OpenJDK11 given that only OpenJDK9/OpenJ9 captured VerifyError. Need to further analyze to see what changes in verifier caused the issue.*

When they attempted to locate the issue by checking the exception table, they found no exception table for the associated method of the mutated class file. Next, they divided the issue into two different checking branches: one was investigating the `simulateStacks` for how it propagated the `uninitalizedThis` (a variable to mark the status of `simulateStacks`) which may or may not be launched in the `mergeStacks` code; the other was comparing the differences in `rtverify.c` for different JVM releases. Finally, by comparing different versions of `rtverify.c`, the developers have identified that a checking mechanism on `uninitializedThis` was disabled in `matchStack()` when creating the `stackmap`. Accordingly, OpenJ9 (1.8.0_232) and OpenJ9 (11.0.5) were confirmed to fail to capture the `VerifyError`.

The buggy instructions of `rtverify.c` are demonstrated in Figure 9. OpenJ9 (1.8.0_232)/OpenJ9 (11.0.5) were allowed to correctly throw `VerifyError` when enabling the checking mechanism on `uninitializedThis` by removing line 1 in Figure 9. However, since such a checking mechanism was designed to prevent a Spring verifier issue [2], it could not be removed simply. Meanwhile, even though the `VerifyError` could be correctly captured on OpenJ9 (9.0.4), its associated `VerifyError` message was rather out-dated. Such issues together deliver a potential demand on upgrading the verification logic of OpenJ9. At last, the developers have stated that they intend to generate a patch to fix all of the exposed issues [4].

*4.4.4 Controversy-Arousing Issue.* In addition to assisting developers in exploring the "deep" bugs, we even triggered an in-depth discussion and revisit to the validity of well-established JVM mechanisms via a potential issue reported by *SJFuzz*.

We have found an issue that OpenJ9 could break the structured locking. In particular, when executing the corresponding

```
if (!verifyData->createdStackMap) { // enable to fix another issue
  if (liveStack->uninitializedThis
  && !targetStack->uninitializedThis) {
    rc = BCV_FAIL;
    goto _finished;
  }
}
```

**Figure 9: OpenJ9 buggy code in rtverify.c.**

```
        ...
66      return0  // return without exitmonitor
        ...
265     monitorenter  // enter the monitor
        ...
709     invokestatic 911 Print.logPrint
712     iload 5
714     iconstm1
715     iadd
716     istore 5
718     iload 5
720     ifle 66  // go to line 66
        ...
```

**Figure 10: The IllegalMonitorStateException issue of OpenJ9.**

mutated class `DirectoryScanner.class`, OpenJDK threw an `IllegalMonitorStateException` because the executing thread accessed a method and executed `entermonitor`, but simply returned without executing `exitmonitor`. However, OpenJ9 did not throw `IllegalMonitorStateException`. Accordingly, we inferred that OpenJ9 allowed returning a method under mismatching `entermonitor` and `exitmonitor` (which broke structured locking) and have reported it to the OpenJ9 developers [3]. Figure 10 refers to the partial class file that exposed this issue.

At first, the developers denied the potential violation of structured locking, i.e., they analyzed our submitted class file and claimed no violation of structured locking. However, during our further investigation, we discovered that while `exitmonitor` has not been executed from line 265 to line 720 in Figure 10, line 720 was executed followed by a `return` instruction (line 66) where `IllegalMonitorStateException` should have been thrown. Correspondingly, the developers reconsidered this issue and finally agreed on the violation of structured locking.

Since the developers still insisted on the legitimacy of their development schemes, they further questioned and inspected the validity of the structured lock mechanism.

> *We may end up with cleaner locking code if we enforced structured locking. This also came up recently in a discussion on how to handle OSR points for inlined synchronized methods. We should investigate the benefits/costs of adopting Structured Locking.*

By tracing back to the JVM specification [18] on the structured locking mechanism, the developers argued that structured locking could be allowed, yet not required. As a result, they considered revoking structured locking to be more as an domain-specific adaptation, rather than a bug, controversially.

In summary, *SJFuzz* is capable of detecting multiple types of "deep" bugs via exposing inter-JVM discrepancies for testing analytics. Furthermore, the bugs detected by *SJFuzz* can be rather tricky to be explored by the existing approaches, e.g., the bug incurred by unresolved JUnit test dependency and the bug that urged developers to trace back to JVM specifications.

## 5 THREATS TO VALIDITY

The threats to external validity mainly lie in the subjects and faults used in our benchmark. To reduce the threats, we determine to select all the possible projects from *Classming*. Moreover, we extend our selections of seeding class files to complicated and popular Java projects such as *Ant*, for evaluating the scalability of our approach.

The threats to internal validity mainly lie in the potential faults in our implementation (including dependent libraries). To reduce such a threat, we apply mature libraries, such as Soot, to implement *SJFuzz*. We also carefully review and test our implemented code and the library code. As a result, we even detected a defect in our adopted Soot version which injected unexpected string into the output class files such that a valid class file was presented as invalid. Correspondingly, we hacked Soot's source code and fixed this issue.

The threats to construct validity mainly lie in the metrics used. To reduce the threats, we leverage various widely used metrics for JVM testing, including the number of discrepancies, as well as the class file diversity and unique bugs found.

## 6 RELATED WORK

**JVM and Compiler Testing.** In addition to the aforementioned *ClassFuzz*, *JavaTailor* and *Classming*, Sirer and Bershad [64] first proposed a grammar-based approach to generate class files by randomly changing a single byte in a seed input which can be hardly applied for deeply testing JVMs. Yoshikawa et al. [75] developed a type system that generates Java class files, which are random, executable, and finite, and then tested them on selected JIT compiler or other Java runtime environments. Freund and Mitchell [45] developed a type system specification for a subset of the bytecode language with type checking and prototype bytecode verifier implementation. Savary et al. [60] derived an abstract model from formal specifications to test the Java byte code verifier. Calvagna and Tramontana [30] proposed an automated conformance testing approach to model JVM as a finite state machine and derive test suites to expose their unexpected behaviors. More recently, Padhye et al. [58] automatically guided QuickCheck-like random input generators to semantically analyze test programs for generating test-oriented Java bytecode. Since the coverage data in Java Just-in-Time compilers (JITs) is deterministic, Wu et al. proposed JITfuzz [71] which leverages the power of coverage guidance and optimization-activation mutators to test JVM JITs. On the other hand, since the coverage is non-deterministic in the whole JVM as mentioned, *SJFuzz* adopts discrepancy and diversity to guide the fuzzing campaign for JVM. Gao et al. [46] incorporated code representation learning and clustering to improve the performance of program-synthesis-based JVM testing (such as JavaTailor).

There are some work on compiler testing [32, 33, 81]. For example, Yang et al. [74] proposed a random mutation-based compiler testing tool for open-source C compilers, which crashed every compiler they tested and found 325 previously unknown bugs in three years. More recently, Cummins et al. [40] developed DeepSmith for accelerating compiler validation via deep learning to model the real-world code structures and generate vast realistic programs to expose compiler bugs. Similarly, Liu et al. [55] automatically generated well-formed C programs to fuzz off-the-shelf C compilers based on generative models.

Compared to the existing JVM and compiler testing approaches either performing worse or requiring extra knowledge in addition to a single seeding class file, *SJFuzz* adopts seed and mutator schedulers based on easy-to-catch runtime discrepancy/diversity information, i.e., acquiring no extra knowledge of bytecode constraints or JVM specifications.

**Seed Scheduling in Fuzzing.** Many coverage-guided fuzzers adopt seed scheduling mechanisms to enhance their effectiveness. AFL [7] typically schedules seeds whenever executing them can increase code coverage. Bohme et al. [29] developed AFLFast to construct a Markov chain model by utilizing coverage feedback, and then scheduled the seeds according to the probability generated from the model for further exploration. They also proposed AFLGo [28] to reach a given program location by scheduling the most related seeds (i.e., the seeds closer to the target location) for mutation. She et al. [63] proposed K-scheduler to schedule the seeds based on all reachable and feasible edges by using the graph analysis of control flow graph. Li et al. [54] proposed Cerebro to schedule seeds based on the code complexity, execution time, and coverage information balanced by an online multi-objective-based algorithm. Chen et al. [36] leveraged the power of different fuzzers by merging seeds generated from them into one corpus and scheduling seeds among various fuzzers. Li et al. [53] proposed Steelix, which utilizes comparison progress information and coverage feedback for scheduling seeds to facilitate the fuzzing efficacy. To bias input generation towards rare branches, Lemieux and Sen [51] proposed FairFuzz to schedule the seeds that hit rare branches for mutation. To facilitate the hybrid fuzzing efficiency, Chen et al. [34] proposed MEUZZ to schedule the seeds between a coverage-guided fuzzer and a concolic execution engine via machine learning. Chen et al. [37] proposed SAVIOR, which schedules the seeds that can reach more sanitizer instrumentation (i.e., a potential buggy point) to expose vulnerabilities of the target program. Zhao et al. [80] proposed DigFuzz to schedule seeds based on the difficulty of their corresponding paths and prioritize them for concolic execution via a Monte-Carlo-based probabilistic path prioritization model. Zhang et al. [78] developed TRUZZ, which schedules the seeds based on the coverage feedback in a newly discovered execution path, i.e., a seed increasing more code coverage has a higher priority to be selected. However, Such existing seed scheduling mechanisms can hardly be used for JVM fuzzing because they are widely guided by code coverage. In this paper, *SJFuzz* adopts diversity and discrepancy as alternative guidance to alleviate the impact of lacking code coverage information for seed scheduling.

**Mutator Scheduling in Fuzzing.** Similar to the above-mentioned seed scheduling mechanisms, many mutator scheduling mechanisms [48, 59, 68, 70, 72, 76, 79, 83] tend to select mutators to increase code coverage during fuzzing. Stephens et al. [65] developed Driller, which selects the symbolic executor to mutate a seed if it fails to increase code coverage under a given time budget. Lyu et al. [56] proposed MOPT, which utilizes Particle Swarm Optimization (PSO) algorithm [42] to find the optimal scheduling probability distribution of mutators via historical code coverage for enhancing fuzzing effectiveness. Fioraldi et al. [44] proposed AFL++ to facilitate fuzzing efficacy by scheduling different mutators from different fuzzers, e.g., mutators from AFL [7] and RedQueen [25]. Wu et al. [69] conducted a study on the havoc fuzzing strategy widely adopted by many coverage-guided fuzzers, and found that applying different mutators leads to different code coverage among various projects. Next, they proposed an improved mutator scheduling mechanism based on a multi-armed bandit algorithm [26] according to the real-time coverage feedback. Xie et al. [73] proposed DeepHunter by scheduling Affine Transformation mutator and Pixel Value Transformation mutator to a given seed via their reference images.

Although failing to exploit code coverage as the existing mutator schedulers, *SJFuzz* still schedules mutators in a lightweight manner by diversifying class file generation via *Monte Carlo method*.

## 7 CONCLUSION

In this paper, we proposed *SJFuzz*, the first fuzzing framework using seed and mutator scheduling for automated JVM differential testing. Specifically, *SJFuzz* employs a discrepancy-guided seed scheduler which retains discrepancy-inducing class files and class files that generate discrepancy-inducing mutants. It also employs a diversity-guided seed scheduler which filters other class files via a coevolutionary mechanism to augment class file diversity for further mutations. Moreover, *SJFuzz* applies a mutator scheduler based on the *Monte Carlo method* to diversify the class file generation. To evaluate the efficacy of *SJFuzz*, we performed an extensive study to compare *SJFuzz* with *Classming*, the state-of-the-art mutation-based JVM fuzzer, on various real-world benchmarks. The results show that overall, *SJFuzz* significantly outperforms *Classming* in terms of exposing inter-JVM discrepancies for JVM differential testing, e.g., *SJFuzz* exposes 8.0 unique discrepancies while *Classming* only exposes 1.1 unique discrepancies averagely on all the studied benchmarks. We also compare *SJFuzz* with the generation-based approach *JavaTailor* in terms of exposing JVM discrepancies. The results also suggest *SJFuzz* outperforms *JavaTailor*, e.g., *SJFuzz* exposes 14.3% more unique discrepancies than *JavaTailor*. To date, we have reported 46 previously unknown potential issues discovered by *SJFuzz* to the JVM developers where 20 were confirmed as bugs and 16 were fixed.

## 8 DATA AVAILABILITY

All the source code, the evaluation details, and the bug reports of this paper are available on our GitHub page [21].

## ACKNOWLEGEMENT

This work is partially supported by the Guangdong Provincial Key Laboratory (Grant No. 2020B121201001) and National Natural Science Foundation of China Grant Nos. 62002256, 62232001. This work is also partially supported by Kuaishou.

# REFERENCES


[1] 2020. JIt Bug. https://github.com/eclipse/openj9/issues/9381.
[2] 2020. Spring Verify Error. https://github.com/eclipse/openj9/issues/5676.
[3] 2020. Structured Locking Issue. https://github.com/eclipse/openj9/issues/9276.
[4] 2020. Verify Bug. https://github.com/eclipse/openj9/issues/9385.
[5] 2021. Motivate Examples. https://github.com/eclipse/openj9/issues/11683.
[6] 2021. Motivate Examples. https://github.com/eclipse/openj9/issues/11684.
[7] 2022. AFL. https://lcamtuf.coredump.cx/afl/.
[8] 2022. Apache Project. https://projects.apache.org/projects.html?language.
[9] 2022. Coevolutionary Algorithm. https://wiki.ece.cmu.edu/ddl/index.php/Coevolutionary_algorithms.
[10] 2022. Definition of Distinct Discrepancy. https://github.com/fuzzy000/SJFuzz/blob/main/src/com/djfuzz/solver/JVMOutputParser.java.
[11] 2022. DragonWell11. https://github.com/alibaba/dragonwell11.
[12] 2022. DragonWell8. https://github.com/alibaba/dragonwell8.
[13] 2022. Gestalt Pattern Matching Distance. https://en.wikipedia.org/wiki/Gestalt_pattern_matching.
[14] 2022. GIJ. https://gcc.gnu.org/onlinedocs/gcc-6.5.0/gcj/.
[15] 2022. HotSpot. http://openjdk.java.net.
[16] 2022. J9. http://www.ibm.com/developerworks/java/jdk.
[17] 2022. Jaro Distance. https://en.wikipedia.org/wiki/Jaro%E2%80%93Winkler_distance.
[18] 2022. The Java Virtual Machine Specification. https://docs.oracle.com/javase/specs/index.html.
[19] 2022. JUnit Official Website. https://junit.org/.
[20] 2022. JVM. https://en.wikipedia.org/wiki/Java_virtual_machine.
[21] 2022. Main Repo for SJFuzz. https://github.com/fuzzy000/SJFuzz.
[22] 2022. Monte Carlo Method. https://en.wikipedia.org/wiki/Monte_Carlo_method.
[23] 2022. Zulu. http://www.azulsystems.com/products/zulu.
[24] 2023. Complement (set theory). https://en.wikipedia.org/wiki/Complement_(set_theory).
[25] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*, Vol. 19. 1–15. https://doi.org/10.14722/ndss.2019.23371
[26] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47, 2 (2002), 235–256.
[27] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. 2002. String matching with metric trees using an approximate distance. In *String Processing and Information Retrieval: 9th International Symposium, SPIRE 2002 Lisbon, Portugal, September 11–13, 2002 Proceedings 9*. Springer, 271–283.
[28] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344. https://doi.org/10.1145/3133956.3134020
[29] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428
[30] A. Calvagna and E. Tramontana. 2013. Automated Conformance Testing of Java Virtual Machines. In *2013 Seventh International Conference on Complex, Intelligent, and Software Intensive Systems*. 547–552. https://doi.org/10.1109/CISIS.2013.99
[31] Junjie Chen, Haoyang Ma, and Lingming Zhang. 2020. Enhanced compiler bug isolation via memoized search. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 78–89. https://doi.org/10.1145/3324884.3416570
[32] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A survey of compiler testing. *Comput. Surveys* 53, 1 (2020), 1–36. https://doi.org/10.1145/3363562
[33] Junjie Chen, Chenyao Suo, Jiajun Jiang, Peiqi Chen, and Xingjian Li. 2023. Compiler Test-Program Generation via Memoized Configuration Search. In *45th IEEE/ACM International Conference on Software Engineering*. IEEE, 2035–2047. https://doi.org/10.1109/ICSE48619.2023.00172
[34] Yaohui Chen, Mansour Ahmadi, Boyu Wang, Long Lu, et al. 2020. {MEUZZ}: Smart Seed Scheduling for Hybrid Fuzzing. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 77–92.
[35] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming Compiler Fuzzers. *SIGPLAN Not.* 48, 6 (jun 2013), 197–208. https://doi.org/10.1145/2499370.2462173
[36] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. {EnFuzz}: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. 1967–1983. https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yuanliang
[37] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1580–1596. https://doi.org/10.1109/SP40000.2020.00002
[38] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1257–1268. https://doi.org/10.1109/ICSE.2019.00127
[39] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 85–99. https://doi.org/10.1145/2908080.2908095
[40] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 95–105. https://doi.org/10.1145/3213846.3213848
[41] Alastair F Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29. https://doi.org/10.1145/3133917
[42] Russell Eberhart and James Kennedy. 1995. A new optimizer using particle swarm theory. In *MHS'95. Proceedings of the sixth international symposium on micro machine and human science*. Ieee, 39–43.
[43] Robert B. Evans and Alberto Savoia. 2007. Differential Testing: A New Approach to Change Detection. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Dubrovnik, Croatia) *(ESEC-FSE '07)*. Association for Computing Machinery, New York, NY, USA, 549–552. https://doi.org/10.1145/1287624.1287707
[44] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi
[45] Stephen N Freund and John C Mitchell. 2003. A type system for the Java bytecode language and verifier. *Journal of Automated Reasoning* 30, 3-4 (2003), 271–321.
[46] Tianchang Gao, Junjie Chen, Yingquan Zhao, Yuqun Zhang, and Lingming Zhang. 2023. Vectorizing Program Ingredients for Better JVM Testing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 526–537. https://doi.org/10.1145/3597926.3598075
[47] Patrice Godefroid. 2020. Fuzzing: Hack, Art, and Science. *Commun. ACM* 63, 2 (jan 2020), 70–76. https://doi.org/10.1145/3363824
[48] Ling Jiang, Hengchen Yuan, Mingyuan Wu, Lingming Zhang, and Yuqun Zhang. 2023. Evaluating and Improving Hybrid Fuzzing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 410–422. https://doi.org/10.1109/ICSE48619.2023.00045
[49] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138. https://doi.org/10.1145/3243734.3243804
[50] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices* 50, 10 (2015), 386–399. https://doi.org/10.1145/2858965.2814319
[51] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 475–485. https://doi.org/10.1145/3238147.3238176
[52] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.
[53] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 627–637. https://doi.org/10.1145/3106237.3106295
[54] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. 2019. Cerebro: Context-Aware Adaptive Fuzzing for Effective Vulnerability Detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 533–544. https://doi.org/10.1145/3338906.3338975
[55] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 1044–1051. https://doi.org/10.1609/aaai.v33i01.33011044
[56] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1949–1966.
[57] Thomas W MacFarland and Jan M Yates. 2016. Mann–whitney u test. In *Introduction to nonparametric statistics for the biological sciences using R*. Springer, 103–132.
[58] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 329–340. https://doi.org/10.1145/3293882.3330576

[59] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 181–198. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[60] A. Savary, M. Frappier, and J. Lanet. 2011. Automatic Generation of Vulnerability Tests for the Java Card Byte Code Verifier. In *2011 Conference on Network and Information Systems Security*. 1–7. https://doi.org/10.1109/SAR-SSI.2011.5931379

[61] Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. 2020. 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*. 737–749. https://doi.org/10.1145/3368089.3409723

[62] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 803–817. https://doi.org/10.1109/SP.2019.00052

[63] D. She, A. Shah, and S. Jana. 2022. Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1558–1558. https://doi.org/10.1109/SP46214.2022.9833761

[64] E. G. Sirer and B. N. Bershad. 1999. Testing Java Virtual Machines, An Experience Report on Automatically Testing Java Virtual Machines. *Proc. Int. Conf. on Software Testing And Review* (1999).

[65] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Krügel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS*.

[66] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press, 13.

[67] Raja Vallee-Rai and Laurie J. Hendren. 1998. Jimple: Simplifying Java Bytecode for Analyses and Transformations.

[68] Mingyuan Wu, Kunqiu Chen, Qi Luo, Jiahong Xiang, Ji Qi, Junjie Chen, Heming Cui, and Yuqun Zhang. 2023. Enhancing Coverage-guided Fuzzing via Phantom Program. In *2023 The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. https://doi.org/10.1145/3611643.3616294

[69] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One Fuzzing Strategy to Rule Them All. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/3510003.3510174

[70] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yuqun Zhang, Guowei Yang, Huixin Ma, Sen Nie, Shi Wu, Heming Cui, and Lingming Zhang. 2022. Evaluating and Improving Neural Program-Smoothing-based Fuzzing. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 847–858. https://doi.org/10.1145/3510003.3510089

[71] Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. [n. d.]. Jitfuzz: Coverage-guided fuzzing for jvm just-in-time compilers. In *Proceedings of the 45th International Conference on Software Engineering, ser. ICSE*, Vol. 23. https://doi.org/10.1109/ICSE48619.2023.00017

[72] Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. 2023. JITfuzz: Coverage-guided Fuzzing for JVM Just-in-Time Compilers. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 56–68. https://doi.org/10.1109/ICSE48619.2023.00017

[73] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 146–157. https://doi.org/10.1145/3293882.3330579

[74] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. https://doi.org/10.1145/3293882.3330579

[75] Takahide Yoshikawa, Kouya Shimura, and Toshihiro Ozawa. 2003. Random program generator for Java JIT compiler test system. In *Third International Conference on Quality Software, 2003. Proceedings.* IEEE, 20–23.

[76] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[77] Kunpeng Zhang, Xi Xiao, Xiaogang Zhu, Ruoxi Sun, Minhui Xue, and Sheng Wen. 2022. Path Transitions Tell More: Optimizing Fuzzing Schedules via Runtime Program States. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1658–1668. https://doi.org/10.1145/3510003.3510063

[78] Kunpeng Zhang, Xi Xiao, Xiaogang Zhu, Ruoxi Sun, Minhui Xue, and Sheng Wen. 2022. Path Transitions Tell More: Optimizing Fuzzing Schedules via Runtime Program States. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/3510003.3510063

[79] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-Based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 132–142. https://doi.org/10.1145/3238147.3238187

[80] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *NDSS*.

[81] Yingquan Zhao, Junjie Chen, Ruifeng Fu, Haojie Ye, and Zan Wang. 2023. Testing the Compiler for a New-Born Programming Language: An Industrial Case Study (Experience Paper). In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 551–563. https://doi.org/10.1145/3597926.3598077

[82] Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-Driven Test Program Synthesis for JVM Testing. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1133–1144. https://doi.org/10.1145/3510003.3510059

[83] Husheng Zhou, Wei Li, Zelun Kong, Junfeng Guo, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. 2020. DeepBillboard: Systematic Physical-World Testing of Autonomous Driving Systems. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 347–358. https://doi.org/10.1145/3377811.3380422


Received 2023-02-02; accepted 2023-07-27