DJFuzz: Discrepancy-driven JVM Fuzzing

Anonymous Author(s)

ABSTRACT

While Java Virtual Machine (JVM) plays a vital role for ensuring correct executions of Java applications, testing JVM 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 inter-JVM execution results for testing analytics. However, their adopted fuzzers are not well guided, making differential testing essentially ineffective. To address such issue, in this paper, we propose *DJFuzz*, the first discrepancy-driven fuzzing framework for automated JVM differential testing. By applying control flow mutators, *DJFuzz* aims to mutate class files to facilitate the exposure of discrepant execution results among different JVMs, i.e., inter-JVM discrepancies. Specifically, *DJFuzz* diversifies the mutation-based class file generation for effectively generating the execution-discrepancy-inducing mutant class files. Also, *DJFuzz* establishes a coevolutionary-algorithm-based filtering mechanism for efficient class file generation. Furthermore, *DJFuzz* also adopts JUnit tests as the entry points for mutation in addition to the traditional `main` methods to augment mutation efficacy. To evaluate *DJFuzz*, we conduct an extensive study on multiple representative real-world JVMs, including OpenJDK, OpenJ9, Alibaba DragonWell, and OracleJDK. The experimental results demonstrate that *DJFuzz* can significantly outperform the state of the art in terms of the inter-JVM discrepancy exposure and the class file diversity, e.g., exposing 3.7×/6.3× more total/unique discrepancies compared with *Classming*. Moreover, *DJFuzz* successfully reported 46 JVM bugs, and 20 of them have been confirmed by the JVM developers.

1 INTRODUCTION

Java Virtual Machine (JVM) refers to the virtual machine that enables a terminal device to interpret and execute Java bytecode compiled from various high-level programming languages, e.g., Java, Scala, and Clojure [11]. 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, to the Metaspace. Then, JVM directly executes the bytecode, or transforms 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 [6], Alibaba's DragonWell [3, 4], IBM's OpenJ9 [7], Azul's Zulu [16], and GNU's GIJ [5], 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, e.g., it can be rather hard to check the correctness of JVM outputs (i.e., the test oracle problem). To address such challenge, prior research works attempt to integrate fuzzing and differential testing for automated JVM testing, i.e., designing fuzzers to generate class files as tests for executing different JVMs such that their discrepant execution results can be used for testing analytics. For instance, *ClassFuzz* [32] fuzzes Java class files by mutating their modifiers or variable types to test the loading, linking, and initialization phases in JVMs. More recently, *Classming* [31] 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.

Though such approaches attempt to improve JVM testing effectiveness, their adopted fuzzers are insufficiently guided and thus can be defective. First, they can hardly exploit the runtime coverage information for fuzzing. Unlike traditional fuzzers which are typically coverage-guided, e.g., iteratively mutating programs for expanding test coverage [21, 25, 27, 43], *ClassFuzz* only collects coverage information for initializing JVMs and *Classming* even exploits no coverage for fuzzing. The main reason is that JVMs are likely to cause non-deterministic coverage at runtime due to their adopted mechanisms, e.g., parallel compilation and on-demand garbage collection [32], which can hardly be leveraged for guiding the fuzzing process. Second, they fail to take advantage of historical execution information. Specifically, while coverage-based fuzzers can easily determine whether retaining seeds for further mutations based on their past execution results at runtime, e.g., collected coverage changes, the existing JVM fuzzers tend to make such decisions simply based on predefined rules [24, 37]. On the other hand, differential testing also can be rather expensive. Specifically, both *ClassFuzz* and *Classming* have to identify one reference JVM for mutating class files first and then use all the resulting mutant class files to execute other JVMs for execution comparison (differential testing). Such process leads to limited usage of the discrepant execution results, i.e., simply for testing analytics rather than runtime guidance for facilitating fuzzing efficacy. Moreover, the execution comparisons relying on one selected reference JVM can also lead to sensitive differential testing outcomes and potentially render inconsistent testing analytics.

To address the aforementioned challenges in testing JVMs, in this paper, we present *DJFuzz* (in our *GitHub* repository [12]), the first end-to-end discrepancy-driven fuzzing framework to generate JVM class files (test cases) which facilitate the exposure of discrepant execution results among different JVMs, i.e., inter-JVM discrepancies, for effective and efficient JVM differential testing via iterative control flow mutations. To this end, *DJFuzz* attempts to iteratively diversify the class file generation by augmenting the overall distances between the seeding and mutant class files. Moreover, *DJFuzz* also filters the seeding class files for efficient mutations. Specifically, since the class files whose mutants can cause inter-JVM discrepancies are likely to generate more such mutants in the future runs, *DJFuzz* retains them for further mutations throughout the rest iterations. On the other hand, for the other class files, *DJFuzz* applies a coevolutionary algorithm to filter them such that the remaining ones can potentially augment the overall class file diversity to facilitate inter-JVM discrepancy exposure for further mutations. Moreover, in addition to the traditional *main-entry* mode (i.e., using `main` method as entry point), we further develop the new *JUnit-entry* mode (i.e., using JUnit tests as entry points) for JVM testing to explore the mutation space more exhaustively.

While *DJFuzz* does not directly exploit coverage information for guiding fuzzing (due to the nondeterminism of JVM systems), it directly adopts inter-JVM discrepancies as guidance, i.e., essential test oracles, to facilitate further discrepancy exposure among multiple JVMs. Moreover, instead of setting a reference JVM in advance and separating test generation and testing stages for traditional JVM differential testing, *DJFuzz* can complete the whole testing process with only one end-to-end run which can be rather efficient. Furthermore, the adopted coevolutionary algorithm of *DJFuzz* can leverage the historical execution information, e.g., the class files whose mutants cause inter-JVM discrepancies, to significantly limit the number of the class file mutations for enhancing the mutation/testing efficiency. At last, unlike the existing JVM testing approaches, i.e., *ClassFuzz* and *Classming*, which adopt primitive mechanisms that fail to backtrack from the previously generated class files, *DJFuzz* involves the existing class files for diversifying the overall class file generation to enhance the test case generation efficacy via a simplistic methodology design.

To evaluate *DJFuzz*, we conduct a set of experiments upon various popular real-world JVMs, e.g., OpenJDK, OpenJ9, DragonWell, and OracleJDK. In particular, we apply *DJFuzz* and state-of-the-art *Classming* to generate class files via seeding class files selected from popular open-source Java projects. Such generated class files are then executed in the studied JVMs to expose their discrepancies for further testing analytics. The experimental results suggest that *DJFuzz* significantly outperforms *Classming* in terms of inter-JVM discrepancy exposure, e.g., exposing 3.7×/6.3× more total/distinct discrepancies on average. Moreover, 46 previously unknown bugs were reported to their corresponding developers after analyzing the discrepancies incurred by *DJFuzz* while none can be detected by *Classming*. As of submission time, 20 bugs have already been confirmed by the developers. In summary, this paper makes the following main contributions:

- **Idea.** To the best of our knowledge, we propose the first discrepancy-driven approach for testing JVM systems, which effectively and efficiently exposes inter-JVM discrepancies.
- **Technique.** We implement our JVM testing approach as a practical system based on Jimple-level mutation via the Soot analysis framework [50]. We also leverage JUnit entry points for JVM testing for the first time to facilitate discrepancy-inducing mutations.
- **Evaluation.** We conduct an extensive set of experiments based on 4 popular JVMs and various real-world benchmark projects. The experimental results demonstrate that *DJFuzz* significantly outperforms state of the art in terms of the inter-JVM discrepancy exposure. Notably, *DJFuzz* reported 46 previously unknown bugs and 20 of them have been confirmed by the developers.

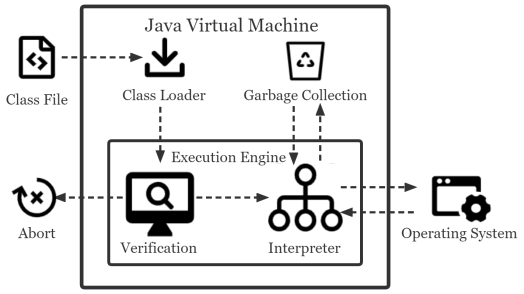


Figure 1: The architecture of JVM.

2 BACKGROUND

In this section, we give an overview on the features of fuzzing, the basic mechanism of Java Virtual Machine (JVM), and the challenges of fuzzing JVMs to illustrate the motivation of our work.

Fuzzing. Fuzzing [23] refers to an automated software testing technique that inputs invalid, 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 [20]. Typically, fuzzers demand guidance. Specifically, a fuzzer iteratively generates mutants and determine whether they can be retained as the seeds for further mutations via its adopted guidance function. While various runtime information can be possibly utilized as guidance, runtime coverage information is widely used as the guidance function [52] for mainstream fuzzers [21, 25, 27, 47, 48, 55] where a mutant is retained as the seed when it explores new code coverage.

Java Virtual Machine. Java Virtual Machine (JVM) is designed for executing programs which can be compiled to JVM byte code. In general, JVM inputs a class file (a compiled file of a target program), resolves its dependencies, executes the whole program, and returns output. Specifically as shown in Figure 1, the `Class Loader` first loads class files into JVM. Next, it sends the instructions to the `Execution Engine` for being executed on the `Operating System`. Subsequently, the `Verification` component verifies whether the instructions are valid or not, and then passes the valid instructions to the `Interpreter`; and aborts the whole execution process otherwise. Note that the instruction could be compiled into the machine code in the `Interpreter` to be executed directly on the corresponding host, which is also known as Just-in-Time compilation [18]. During the entire class file life cycle, the `Garbage Collection` can be activated in any moment to recycle memory for efficient resource management.

Challenges of fuzzing JVMs. Since Just-in-Time compilation in `Interpreter` and `Garbage Collection` can be spontaneous. For instance, it can compile random number of methods simultaneously, which renders the coverage information collected by executing JVMs somehow non-deterministic, i.e., one mutant can incur different code coverage in different runs. Hence collecting such coverage information results in a potential balloon seed corpus with a significant number of redundant mutants and thus seriously compromises fuzzing efficiency. Therefore, existing approaches fuzz

JVMs with limited guidance from runtime/historical information, e.g., *ClassFuzz* only collects code coverage from the `Verification` component and *Classming* even enables no guidance from executing JVMs. As a result, to facilitate the efficacy of fuzzing JVMs, it is essential to propose a new guidance mechanism from the JVM runtime information.

3 THE APPROACH OF *DJFUZZ*

In this section, we propose *DJFuzz*, a discrepancy-driven mutation-based JVM class file fuzzer to effectively and efficiently generate class files for automated JVM differential testing. The framework of *DJFuzz* is demonstrated in Figure 2. Note that *DJFuzz* enables iterative mutation-based class file generation. In particular, given a seeding class file, *DJFuzz* adopts the control flow mutation strategy to generate its mutant class file (Section 3.1). Accordingly, for each iteration, *DJFuzz* generates such mutants for all the seeding class files (Section 3.2) and filters the resulting class file collection (Section 3.3) under the discrepancy guidance for further iterations.

DJFuzz adopts the following assumptions for launching its discrepancy guidance mechanism. On one hand, for one class file, if running its mutant can cause inter-JVM discrepancies, such class file is likely to generate more discrepancy-inducing mutants than other class files since presumably it enables a stronger connection with JVM bugs than other class files. Therefore, such 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 to facilitate the discrepancy-driven class file generation. On the other hand, for the class file whose mutant cannot instantly expose inter-JVM discrepancies, it does not necessarily suggest that it cannot generate discrepancy-inducing mutants in the future iterative executions, i.e., leveraging such class file can also possibly advance the inter-JVM discrepancy exposure. To this end, such class file can be characterized as *optional*. Note that differential testing usually enables vast space for generating test cases while the overall class files to cause the underlying discrepancies between sophisticated JVMs can be rather limited, i.e., the optional class files can significantly outnumber the primary class files. Therefore, it is also essential to efficiently explore the input space via diversifying the generation for both primary and optional class files. To summarize, a discrepancy-guided class file generation mechanism essentially demands both sufficiently and efficiently exploring the discrepancy-driven class file generation space which can intuitively be realized by diversifying the class file generation.

Algorithm 1 demonstrates the details of *DJFuzz* which is initialized by adding one `seedingClass` into the queue and assigning the `seedingClass` to be optional (lines 4 to 5). Under each iterative execution (line 6), *DJFuzz* investigates each class file in the queue to derive and mutate its method via a control flow mutator for obtaining its mutant class file to facilitate the overall class file diversity (lines 8 to 10). A class file can be identified whether to be primary (including the discrepancy-inducing and valid mutants) after running on the adopted JVMs (lines 13 to 17). Meanwhile, for any valid mutant class file, *DJFuzz* updates its distance to its seeding class file and derives its mutation strategy accordingly (lines 18 to 20). At last, all the class files in the `queue` and their mutant class files are combined where all the primary class files and only the optional class files which can diversify their overall distances can be retained for future mutations (lines 24 to 25). Such iterations are terminated after hitting the `budget` with all the inter-JVM discrepancies exposed during the iterative executions. Note that *DJFuzz* only enables valid class files for mutations because usually mutating an invalid class file tends to cause the existing unexpected program behaviors other than exposing unexplored inter-JVM discrepancies.

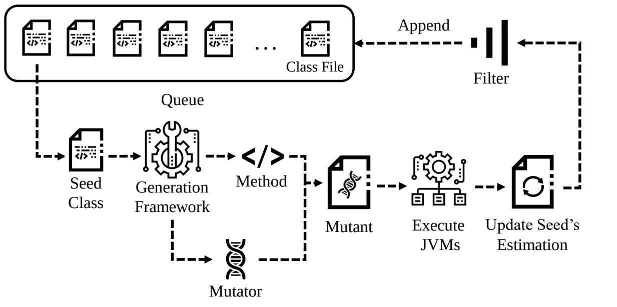


Figure 2: The framework of *DJFuzz*.

To illustrate, our JVM testing framework is "discrepancy-driven" because (1) we retain the class file and its mutant which can cause inter-JVM discrepancies as primary class file for further mutations, and (2) we filter the rest optional class files for further mutations to facilitate inter-JVM discrepancies. Moreover, traditional JVM differential testing implementations, e.g., *ClassFuzz* and *Classming*, can be rather inefficient due to their separated test case generation and differential testing stages, i.e., generating all the test cases first and then using them for differentially testing JVMs. *DJFuzz*, on the other hand, can integrate test generation and testing stages end-to-end, i.e., under each iteration, *DJFuzz* runs the seeding class files upon the adopted JVMs for automatically collecting their discrepancies which can be instantly used for subsequent test generation. To this end, *DJFuzz* can be potentially more efficient.

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 vulnerabilities in their "deep" execution stages [29, 31, 35, 39]. Following such prior work (and also for a fair comparison with them), *DJFuzz* also adopts such control flow mutation with representative mutators. Specifically in source code level, *DJFuzz* randomly selects two original instructions, and creates a directed transition between them. If such transition is a loop, the corresponding iteration will be limited to 5 times. Correspondingly, *DJFuzz* implements the mutators with the Jimple-level instructions `goto`, `lookupswitch`, and `return` provided in *Soot* [50].

DJFuzz iteratively selects random positions from a given method to apply the control-flow mutators. Specifically, *DJFuzz* establishes an *InstructionCollection* list which contains the instructions executed by the adopted JVMs in terms of their execution order. Next, *DJFuzz* selects and inserts a control flow mutator into a random *InstructionCollection* spot under each iteration.

Algorithm 1 Discrepancy-driven Fuzzing

```
    Input: seedingClass, budget, bound
    Output: queue
 1: function DISCREPANCY_GUIDANCE
 2:     iteration ← 0
 3:     queue ← list()
 4:     queue.add(seedingClass)
 5:     seedingClass.primary ← False
 6:     while iteration < budget do
 7:         children ← list()
 8:         for class in queue do
 9:             method ← selectMethod()
10:             mutantClass ← SELECT_MUTATION(class, method)
11:             mutantClass.primary ← False
12:             iteration ← iteration + 1
13:             runJVMs(mutantClass)
14:             if mutantClass incurs new DISCREPANCIES then
15:                 class.primary ← True
16:                 if mutantClass is VALID then
17:                     mutantClass.primary ← True
18:             if mutantClass is VALID then
19:                 distance ← Levenshtein(class, mutantClass)
20:                 updateEstimation(class, distance)
21:                 children.add(mutantClass)
22:             else
23:                 updateEstimation(class, -1)
24:         merge(queue, children)
25:         queue ← FILTER_OPTIONAL_CLASSES(queue, bound)
26:     return queue
```

3.2 Mutant Class File Generation

To diversify the class file generation, for a given class file, *DJFuzz* first selects its method to be mutated and then determines its mutator such that the resulting mutant class file can potentially facilitate the overall edit-distance-based class file diversity.

3.2.1 Method Selection. Given a class file, *DJFuzz* selects a method to be mutated based on the following intuitions: (1) JVM class files enable hierarchical semantics among instructions. The *entry methods*, i.e., methods within the seeding class files directly or indirectly invoked by the corresponding main methods (including the main method itself) or JUnit tests, are likely to represent the overall semantics than other instructions; and (2) the methods with more instructions indicate exponentially more mutation entry points,i.e., a method with n instructions results in 2^n potential executions after mutations because any of its instructions can be mutated to be either executed or non-executed. Therefore, mutating such methods are more likely to expose inter-JVM discrepancies.

For Intuition (1), we determine to bound our mutation scope on the entry methods for efficiency. For Intuition (2), we assign method m_i with its selection probability $Prob_{m_i}$ as in Equation 1, where $Prob_{m_i}$ is computed as the ratio of the number of instructions of m_i over the total number of the instructions of all the entry methods. As a result, a seeding class file selects an entry method based on the derived selection probability:

$$Prob_{m_i} = \frac{instructions_{m_i}}{\sum_{j=1}^{total} instructions_{m_j}} \tag{1}$$

3.2.2 Mutator Selection. Since it is challenging to derive the exact diversity of the overall class files on the fly, *DJFuzz* selects mutators to diversify the seeding and mutant class files under each iteration to approximate the overall class file diversity instead. In particular, *DJFuzz* first applies the edit distance, i.e., Levenshtein Distance [41] in this paper, to delineate the diversity between a pair of class files. Accordingly, *DJFuzz* establishes a *deterministic mutator selection strategy* for estimating the mutator that can optimize the seed-mutant distance. Meanwhile, *DJFuzz* also develops a *random mutator selection strategy* to prevent the potential local optimization that can derive sub-optimal mutators caused by the *deterministic mutator selection strategy*. As a result, *DJFuzz* derives a mutator for a given class/method by combining the two strategies.

Edit distance-based diversity measure. To accurately reflect the fine-grained differences between class files, an ideal metric is expected to reflect their instruction-by-instruction comparisons. However, it can be cost-inefficient. In this paper, we determine to adopt the instructions from the entry methods, namely *EntryInstruction*, as the representative instructions for efficiently measuring 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., loops. Eventually, we characterize the diversity measurement between a pair of class files as deriving differences between their associated *EntryInstructions* which are further placed in *InstructionCollection* for mutation.

In this paper, *DJFuzz* applies edit distance, i.e., Levenshtein Distance [41], a metric that is 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. To be specific, *DJFuzz* 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 seeding and mutant class files can be modeled legislate 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.

Deterministic mutator selection strategy. 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, *DJFuzz* adopts the *Monte Carlo method* [19] to develop the *deterministic mutator selection strategy*, where given a selected method m_p, *DJFuzz* 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 2. Such value function can reflect the resulting diversity of the overall class files under the mutation, i.e., the cumulative diversity expectation between the seeding and mutant class files under all the iterations.

Algorithm 2 Mutator Selection

```
    Input : class, method
    Output: mutantClass
1:  function SELECT_MUTATION
2:      rand ← Random()
3:      if rand < explorationRate then
4:          mutator ← selectRandomMutator(class, method)
5:          return mutatedClass ← mutator.mutate(class)
6:      bestMutator ← selectDeterministicMutator(method)
7:      return mutatedClass ← bestMutator.mutate(class)
```

$$V(c_i, a_j, m_p) = \mathbb{E}[\frac{1}{N}\sum_{k=0}^{N} Distance_k] \quad (2)$$

Here *Distance* refers to the Levenshtein Distance between the seeding class file c_i and its mutant class file by applying mutator a_j. $Distance_k$ refers to their Levenshtein Distance in the k_{th} iteration which can be dynamically updated since the mutation spot is randomly selected in m_p under each iteration. $\mathbb{E}$ refers to expectation. It can be easily derived that $V(c_i, a_j, m_p)$ for c_i can be incrementally updated and inefficient to be directly computed. Therefore, we further enable dynamic updates on $V(c_i, a_j, m_p)$ as presented in Equation 3 to approximate its value, where α is a constant. Note when one class file c_i fails to generate a valid class, its *Distance* is set to -1.

$$V(c_i, a_j, m_p) = V(c_i, a_j, m_p) + \alpha(Distance - V(c_i, a_j, m_p)) \quad (3)$$

As a result, we select a mutator a_j corresponding to the largest $V(c_i, a_j, m_p)$ for c_i. Typically, *DJFuzz* 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 20 of Algorithm 1).

Random mutator selection strategy. 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 issue, *DJFuzz* further leverages a *random mutator selection strategy* to reduce its possibility to select a sub-optimal mutator under early-terminated executions by randomly selecting one mutator for class file generation under the ongoing iteration. As a result, by properly combining such strategy with the *deterministic mutator selection strategy*, 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 selection strategy is presented in Algorithm 2. We first set an `explorationRate` and generate a random value for comparison (line 2). Next, if such random value is less than the `explorationRate`, *DJFuzz* chooses the *random mutator selection strategy* that returns a random mutator under the ongoing iteration (lines 3 to 5). Otherwise, *DJFuzz* derives the mutator by applying the *deterministic mutator selection strategy* (lines 6 to 7).

3.3 Class File Filtering

As mentioned, it is essential to limit the number of class files for efficient mutation-based generation. To this end, *DJFuzz* filters the collected class files after all the mutations under each iteration. Typically, *DJFuzz* retains all the primary class files for further iterative mutations. On the other hand, *DJFuzz* filters the optional class files such that the remaining ones can be mutated to potentially augment the overall class file diversity. To this end, *DJFuzz* adopts the coevolutionary algorithm [22] because it can 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, the fitness computation details are presented at Algorithm 3. for one optional class file, *DJFuzz* 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, *DJFuzz* selects the top-N corresponding optional class files for further mutation-based generation, where N is predefined.

Algorithm 3 Fitness Calculation

```
    Input: orignalQueue
1:  function CALCULATE_DIVERSE_FITNESS
2:      classes ← list()
3:      for class in orignalQueue do
4:          if class.primary is False then
5:              classes.add(class)
6:      for target in classes do
7:          distance ← 0
8:          for class in classes do
9:              if class equals target then
10:                 continue
11:             distance ← distance + Levenshtein(class, target)
12:         target.setFitness(distance / length(classes))
```

In summary, *DJFuzz* adopts discrepancy-driven framework based on control flow mutations. Specifically, under each iterative execution, *DJFuzz* adopts mutation strategy to diversify the class file generation via *Monte Carlo method*. It also applies the coevolutionary algorithm to filter seeding class files for enhancing the class file generation efficiency and facilitating class file diversity.

4 EVALUATION

We conduct a set of experiments on various popular JVMs. Overall, we aim to compare *DJFuzz* with state of the art in terms of their resulting inter-JVM discrepancies, the class file generation efficiency, and the reported bugs/defects by answering the following research questions:

- **RQ1:** Is *DJFuzz* effective in exposing inter-JVM discrepancies?
- **RQ2:** Is the discrepancy guidance mechanism effective?

Furthermore, we report and analyze the bugs detected by *DJFuzz* with all the evaluation details presented in our GitHub page [12].

4.1 Benchmark Construction

We adopt multiple widely-used real-world JVMs, i.e., OpenJDK, OpenJ9, DragonWell, and OracleJDK, for running *DJFuzz* to expose their discrepancies. We also adopt state-of-the-art *Classming* as the baseline for comparison as it outperformed other existing approaches for JVM differential testing [31]. Specifically, for a fair comparison with *DJFuzz* which integrates differential testing and test generation stages, we also run *Classming* on all studied JVMs in parallel to complete the differential testing once and for all.

To launch *DJFuzz*, we adopt 26 class files 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 [1]) and `JUnit` [10] (a widely used unit testing framework) to expand our benchmark diversity.

Note that while the existing approaches, e.g., *Classming*, 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 [31, 32], the new *JUnit-entry* mode, on the other hand, refers to mutating other entry methods associated with JUnit tests.

JUnit-entry can benefit the class file generation with 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.

We have observed that *DJFuzz* has rather stable performance across different configurations; the detailed experimental results can be found on our GitHub page [12] due to space limit. Similar as prior work [32], all benchmarks are executed by all the studied approaches for 24 hours to generate class files to reflect a large enough testing `budget`.

4.3 Result Analysis

4.3.1 RQ1: The inter-JVM discrepancy exposure effectiveness of DJFuzz.

The inter-JVM discrepancy results (both the total number and the distinct types) after executing the generated class files are presented in Table 1 where the distinct discrepancy type refers to the unique inter-JVM discrepancies including the unique exceptions/errors exposed as presented in [2]. For instance, under benchmark `Jython.class`, *DJFuzz* can expose a total of 863 execution discrepancies with 9 distinct types.

Table 1: Discrepancies exposed by *DJFuzz* and *Classming*.

Project	Benchmark	*DJFuzz*		*Classming*	
		Total	**Distinct**	**Total**	**Distinct**
Eclipse	EclipseStarter.class	2756	11	247	1
Fop	Fop.class	25	2	1	1
Jython	Jython.class	863	9	14	1
Sunflow	Benchmark.class	1854	10	337	1
Ant	Launcher.class	2594	12	494	1
Ivy	Main.class	5205	20	1514	3
Fop (JUnit-entry)	FopConfParser.class	1576	10	84	2
	FopFactoryBuilder.class	1556	11	337	3
	ResourceResolverFactory.class	232	4	25	1
	FontFileReader.class	792	5	111	1
Jython (JUnit-entry)	PyByteArray.class	2278	9	843	2
	PyFloat.class	961	4	244	1
	PySystemState.class	251	5	59	1
	PyTuple.class	1654	9	327	2
Ant (JUnit-entry)	AntClassLoader.class	291	12	7	1
	DirectoryScanner.class	82	10	3	1
	Project.class	53	3	0	0
	Locator.class	1715	10	437	1
Ivy (JUnit-entry)	ResolveReport.class	213	5	0	0
	ApacheURLLister.class	646	7	211	1
	Configurator.class	544	8	64	1
	IvyEventFilter.class	1286	10	401	2
JUnit (Junit-entry)	RuleChain.class	1047	15	259	1
	TestWatcher.class	781	3	237	1
	ErrorReportingRunner.class	2051	11	535	1
	Money.class	979	12	92	1
Average		1241.7	8.7	264.7	1.2

We can observe that overall, *DJFuzz* can significantly outperform *Classming* in terms of the inter-JVM discrepancy exposure. To be specific, *DJFuzz* can expose averagely 1241.7 total discrepancies with 8.7 distinct types, while *Classming* can expose averagely 264.7 total discrepancies with 1.2 distinct types, i.e., over 3.7×/6.3× more total/distinct discrepancies. Moreover, we can further find that for all the adopted benchmark projects, *DJFuzz* can significantly outperform *Classming* in terms of both the total and distinct discrepancy exposure. Such results can reflect that our adopted discrepancy guidance mechanism, including diversifying and filtering class file generation, can be quite effective.

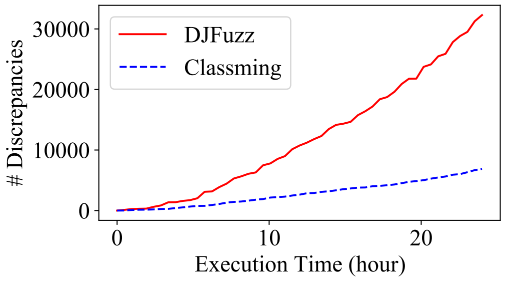


Figure 3: *DJFuzz*/*Classming* efficiency in 24 hours.

We further investigate the impact of the execution time on discrepancy exposure by *DJFuzz* and *Classming*. Figure 3 shows how the total exposed discrepancies on all the benchmarks by the two approaches vary over time. We can observe that although we enhanced the differential testing efficiency by running JVMs in parallel for *Classming* (as in Section 4.1), *DJFuzz* can still significantly outperform *Classming* by exposing 32285 vs. 6883 discrepancies in total. We can also observe that *DJFuzz* consistently outperforms

Classming in finding JVM discrepancies all the time before terminating the executions. Such result can further indicate the power of the discrepancy guidance mechanism of *DJFuzz*. Note that we also evaluate their efficiency, e.g., trends with and without applying the "discrepancy-driven" concept. Due to the page limit, such results are presented in our GitHub page [12].

Finding 1: *DJFuzz is effective by exposing 3.67× more discrepancies than Classming (32285 vs. 6883) under the same evaluation setups.*

We also investigate the discrepancies exposed by our adopted entry modes for class file mutations: *main-entry* and *JUnit-entry*. Specifically, *DJFuzz* can significantly outperform *Classming* under both the entry modes, i.e., *DJFuzz* can expose 64 distinct discrepancies in 6 *main-entry* benchmarks and 163 distinct discrepancies in 20 *JUnit-entry* benchmarks, while *Classming* can only expose 8 and 24 distinct discrepancies under such two entry modes respectively. Additionally, for the projects which enable both *main-entry* and *JUnit-entry* (i.e., `Fop`, `Jython`, `Ant`, and `Ivy`), *DJFuzz* exposes 122 distinct discrepancies in total under *JUnit-entry* and 43 under *main-entry*. Such result indicates the effectiveness of our newly proposed *JUnit-entry* mode for JVM testing. We highly encourage future researchers/practitioners to look into such *JUnit-entry* mode for advancing JVM testing.

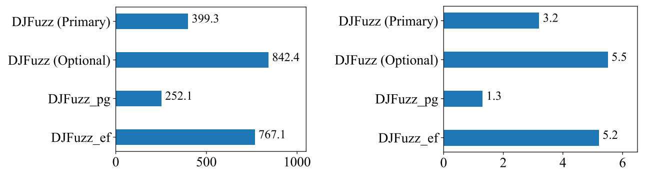
Finding 2: *JUnit-entry is more effective than main-entry in exposing inter-JVM exposures.*

4.3.2 RQ2: Effectiveness of the discrepancy guidance mechanism.

To investigate the effectiveness of the adopted discrepancy guidance mechanism of *DJFuzz*, we record the number of discrepancies exposed by the primary and optional class files of the original *DJFuzz* approach, denoted as *DJFuzz*(primary) and *DJFuzz*(optional), respectively. Furthermore, we also build the following two variant techniques of the original *DJFuzz*: (1) *DJFuzz*$_{pg}$, which only allows primary class files for the mutation-based test case generation as in Section 3.2, and (2) *DJFuzz*$_{ef}$, which equally filters the class files as in Section 3.3 regardless whether they are primary or optional. Note that *DJFuzz*$_{pg}$ is launched via an initial primary class file. Therefore, *DJFuzz*$_{pg}$ retains the initial class file (which may not be a primary class file) for further mutations until it explores a primary class file.

Effectiveness of *DJFuzz*$_{pg}$. In general, we can observe from Figure 4 that *DJFuzz*$_{pg}$ can be effective by exposing 252.1 discrepancies with 1.3 distinct types on average. Interestingly, only *DJFuzz*$_{pg}$ itself can enable quite close performance with state-of-the-art *Classming* (264.7 discrepancies with 1.2 distinct types on average as in Table 1). Such results can indicate the effectiveness of our "discrepancy-driven" intuition, i.e., exploiting the power of discrepancy-inducing class files can advance JVM differential testing.

Effectiveness of *DJFuzz*$_{ef}$. We can also observe from Figure 4 that *DJFuzz*$_{ef}$ can be effective by exposing 767.1 total discrepancies with 5.2 distinct types. Since *DJFuzz*$_{ef}$ essentially refers to equally filtering all the class files to facilitate their seed-mutant distance, i.e., diversity, for further mutation-based class file generation, the fact



(a) All discrepancies. (b) Distinct discrepancies.

Figure 4: Average number of discrepancies found by *DJFuzz*, *DJFuzz*$_{pg}$ and *DJFuzz*$_{ef}$ in all benchmarks.

that *DJFuzz*$_{ef}$ outperforms *DJFuzz*$_{pg}$ implies that such diversity-oriented class file filtering mechanism makes a vital contribution to exposing inter-JVM discrepancies.

Effectiveness of integrating *DJFuzz*$_{pg}$ and *DJFuzz*$_{ef}$. Interestingly, Figure 4 demonstrates that by integrating *DJFuzz*$_{pg}$ and *DJFuzz*$_{ef}$, i.e., applying the original *DJFuzz*, mutating primary class files can incur significantly more inter-JVM discrepancies, i.e., 399.3 vs. 252.1 discrepancies with 3.2 vs. 1.3 distinct types between *DJFuzz*(primary) and *DJFuzz*$_{pg}$. Such results can 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 overall chances to expose discrepancies can be augmented. On the other hand, the fact that *DJFuzz*(optional) outperforms *DJFuzz*$_{ef}$ 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, *DJFuzz*$_{pg}$/*DJFuzz*$_{ef}$ can mutually advance each other to optimize the performance of *DJFuzz*.

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

The previous findings of the effectiveness of different *DJFuzz* 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 the class file diversity in terms of the average seed-mutant Levenshtein Distance of the collected class files.

The diversity results of the class file generation are presented at Figure 5. We can observe that *DJFuzz* incurs much larger average seed-mutant Levenshtein Distance compared with *Classming*, i.e., overall 25.0× larger and 32.1×/23.6× larger under *main-entry*/*JUnit-entry*. It can be inferred that *Classming* tends to generate similar mutants, which also indicates the effectiveness of *DJFuzz*'s diversity-oriented class file generation mechanism.

We further attempt to infer the possible reasons behind the diversity performance difference between *DJFuzz* and *Classming* in terms of the seed-mutant Levenshtein Distance. Assume a mutated

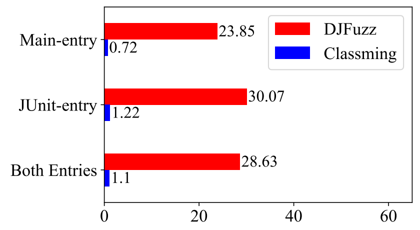


Figure 5: Average seed-mutant levenshtein distance.

```
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*.

Table 2: Issues found by *DJFuzz*.

JVMs	# Issues Reported				# Issues 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

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 error can hinder the detection of “deep” bugs, e.g., bugs incurred in execution engine. However, under such 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, *DJFuzz* is free from such constraints because it can diversify seeding optional class files with best effort. Moreover, even when a seeding 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 4: *Diversifying class file generation is advanced in testing the “deep” bugs in execution engine by retaining sufficient valid class files.*

4.4 Bug Report and Discussion

We manually analyze all the collected discrepancies to derive potential defects. Note that in this paper, we define a defect as an error or an unexpected behavior for a specific JVM version. As a result, we report 46 potential defects from the discrepancies found by *DJFuzz*, as in Table 2, to their corresponding developers. As of today, 20 of them have been confirmed by developers. We select and present some example bug reports as follows.

```
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 7: Runtime inconsistency bug in OpenJ9.

4.4.1 Resource retrieval bug. We have reported an OpenJDK bug on retrieving JAR information which has been confirmed by the OpenJDK developers and assigned with a bug ID `JDK-8244083`. Such bug was exposed by the execution discrepancy between OpenJ9 and OpenJDK. To be specific, 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 7. 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 discrepancy may be caused by the unresolved dependency between JUnit tests and reported it to the corresponding developers [9].

To tackle such 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:

> *It looks like 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 if (!verifyData->createdStackMap) { // enable to fix another issue
2   if (liveStack->uninitializedThis
3   && !targetStack->uninitializedThis) {
4     rc = BCV_FAIL;
5     goto _finished;
6   }
7 }
```

Figure 8: OpenJ9 buggy code in rtverify.c.

(1.8.0_232) and OpenJ9 (11.0.5) wrongly took it as a valid class file 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 8. 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 8. However, since such checking mechanism was designed to prevent a Spring verifier issue [13], 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 [15].

4.4.4 JVM crashes. *DJFuzz* discovered a JVM crash in OpenJ9(11.0.9) by running generated classes from `RuleChain.class`. In particular, a segment fault occurred in `inlineFastObjectMonitorEnter()` for OpenJ9(11.0.9) as shown in Figure 9. We have reported this issue to the corresponding developers [17].

Developers performed a preliminary analysis and found that when accessing `J9Class lockOffset (0xd8)`, `rax` did not contain a valid `J9Class` address after executing the gdb command `info reg rax` from the decompiled assembly code in Figure 10. Likely

```
inlineFastObjectMonitorEnter(J9VMThread *currentThread, j9object_t object)
{
 bool locked = false;
 if (LN_HAS_LOCKWORD(currentThread, object) {
     ...
 }
 ...
}
```

Figure 9: Crash in ObjectMonitor.hpp.

```
...
xor    %al,%al
mov    0xd8(%rax),%rdx
test   %rdx,%rdx
js     0x7fa7f7adcd14
...
```

Figure 10: Assembly code from ObjectMonitor.hpp.

```
        ...
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 11: The IllegalMonitorStateException issue of OpenJ9.

the object was invalid already. As a result, they decided to further analyze the root cause with the following feedback:

> *This will hopefully also tell us when the object went bad (though this is not guaranteed since the GC may move the object and not update some of the stack frames: the forwarder constructors never read the object again after they call).*

4.4.5 Bug under discussion (unconfirmed yet). 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 defect reported by *DJFuzz*.

We have found an issue that OpenJ9 could break the structured locking. In particular, when executing the corresponding mutated class `DirectoryScanner.class`, OpenJDK threw an `IllegalMonitorStateException` because the executing thread accessed a method and executed `entermonitor`, but simply returned without executing `exitmonitor`. On the other hand, 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 [14]. Figure 11 refers to the partial class file that exposed such 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 11, 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 [8] 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, *DJFuzz* is capable of detecting multiple types of "deep" bugs via exposing inter-JVM discrepancies for testing analytics. Furthermore, the bugs detected by *DJFuzz* 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 threat, we apply mature libraries, such as Soot, to implement *DJFuzz*. 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 Testing. In addition to the aforementioned *ClassFuzz* and *Classming*, Sirer et al. [49] first proposed 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. [54] developed a test system that generates Java class files which are random, executable, and finite, and then tested them on selected JIT compiler and other Java runtime environments. Freund et al. [36] developed a type system specification for a subset of the bytecode language with a type checking algorithm and prototype bytecode verifier implementation. Savary et al. [46] derived an abstract model from formal specifications to test the Java byte code verifier. Calvagna et al. [28] 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. [44] automatically guided QuickCheck-like random input generators to semantically analyze test programs for generating test-oriented Java bytecode.

Our approach *DJFuzz* adopts control flow mutation for establishing a discrepancy-guided framework to efficiently diversify the test generation. Compared with state-of-the-art *Classming* and other approaches (which either perform worse or require additional knowledge), *DJFuzz* constructs the test generation as a black box process and thus does not acquire extra knowledge of bytecode constraints or JVM specifications.

Compiler Testing and Fuzzing. Compiler testing and fuzzing have been extensively studied for decades; please refer to a recent survey for more details [30]. Yang et al. [53] proposed a random test generation tool for open-source C compilers, which crashed every compiler they tested and found 325 previously unknown bugs in three years. A recent work by Le et al. [38] leveraged equivalence modulo inputs (EMI) to validate different C compilers. Furthermore, they also introduced a guided and advanced mutation strategy based on Bayesian optimization for compiler testing [40]. Chowdhury et al. [33] developed novel techniques for EMI-based mutation of CPS models, including dealing with language features that do not exist in procedural languages. More recently, Cummins et al. developed DeepSmith [34] 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. [42] automatically generated well-formed C programs to fuzz off-the-shelf C compilers based on generative models.

Besides compiler testing, the broad area of fuzz testing also involves structurally complex test input generation. Wang et al. [51] leveraged the knowledge in the vast amount of existing samples to generate well-distributed seed inputs for fuzzing programs that process highly-structured inputs. Bohme et al. [26] proposed AFLGO based on AFL [21] which generates inputs to fuzz programs by reaching their program locations efficiently. By observing that many fuzzing tools follow a common solution pattern, Padhye et al. [45] developed domain-specific fuzzing tools to augment fuzzing by saving intermediate inputs. Such existing fuzzers can hardly be used for JVM fuzzing because they are widely guided by coverage which can be non-deterministic and thus hard to guide JVM fuzzing.

7 CONCLUSION

In this paper, we have proposed *DJFuzz*, first discrepancy-driven fuzzing framework for automated JVM differential testing. Specifically, *DJFuzz* adopts the *Monte Carlo method* to diversify the class file generation. Moreover, *DJFuzz* also adopts the coevoloutionary mechanism for augmenting the class file generation efficiency. To evaluate the efficacy of *DJFuzz*, we performed an extensive study to compare *DJFuzz* with state-of-the-art *Classming* on various real-world benchmarks. The results show that on average, *DJFuzz* exposes 8.7 unique discrepancies while *Classming* only exposes 1.2 unique discrepancies. To date, we have reported 46 previously unknown bugs discovered by *DJFuzz* to the JVM developers and 20 of them have been confirmed.

REFERENCES

[1] 2020. Apache Project. https://projects.apache.org/projects.html?language.
[2] 2020. Definition of Distinct Discrepancy. https://github.com/fuzzy000/DJFuzz/blob/main/src/com/djfuzz/solver/JVMOutputParser.java.
[3] 2020. DragonWell11. https://github.com/alibaba/dragonwell11.
[4] 2020. DragonWell8. https://github.com/alibaba/dragonwell8.
[5] 2020. GIJ. https://gcc.gnu.org/onlinedocs/gcc-6.5.0/gcj/.
[6] 2020. HotSpot. http://openjdk.java.net.
[7] 2020. J9. http://www.ibm.com/developerworks/java/jdk.
[8] 2020. The Java Virtual Machine Specification. https://docs.oracle.com/javase/specs/index.html.
[9] 2020. JIt Bug. https://github.com/eclipse/openj9/issues/9381.
[10] 2020. JUnit Official Website. https://junit.org/.
[11] 2020. JVM. https://en.wikipedia.org/wiki/Java_virtual_machine.
[12] 2020. Main Repo for DJFuzz. https://github.com/fuzzy000/DJFuzz.
[13] 2020. Spring Verify Error. https://github.com/eclipse/openj9/issues/5676.
[14] 2020. Structured Locking Issue. https://github.com/eclipse/openj9/issues/9276.
[15] 2020. Verify Bug. https://github.com/eclipse/openj9/issues/9385.
[16] 2020. Zulu. http://www.azulsystems.com/products/zulu.
[17] 2021. Crash In OpenJ9(11.0.9). https://github.com/eclipse/openj9/issues/11725.
[18] 2021. Just-in-time compilation. https://en.wikipedia.org/wiki/Just-in-time_compilation.
[19] 2021. Monte Carlo Method. https://en.wikipedia.org/wiki/Monte_Carlo_method.
[20] 2021. Real-world Vulnerabilities Exposed by Fuzzing. https://github.com/mrash/afl-cve.
[21] 2022. AFL. https://lcamtuf.coredump.cx/afl/.
[22] 2022. Coevolutionary Algorithm. https://wiki.ece.cmu.edu/ddl/index.php/Coevolutionary_algorithms.
[23] 2022. Fuzzing. https://en.wikipedia.org/wiki/Fuzzing.
[24] Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I Jordan. 2003. An introduction to MCMC for machine learning. *Machine learning* 50, 1 (2003), 5–43.
[25] 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.
[26] 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.
[27] 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
[28] 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.
[29] 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.
[30] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1, Article 4 (Feb. 2020), 36 pages. https://doi.org/10.1145/3363562
[31] 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.
[32] 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.
[33] Shafiul Azam Chowdhury, Sohil Lal Shrestha, Taylor T Johnson, and Christoph Csallner. [n.d.]. SLEMI: Finding Simulink Compiler Bugs through Equivalence Modulo Input (EMI). ([n. d.]).
[34] 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.
[35] 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.
[36] 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.
[37] Daniel Huang, Jean-Baptiste Tristan, and Greg Morrisett. 2017. Compiling Markov chain Monte Carlo algorithms for probabilistic modeling. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 111–125.
[38] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216–226. https://doi.org/10.1145/2594291.2594334
[39] 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.
[40] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 386–399. https://doi.org/10.1145/2814270.2814319
[41] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.
[42] 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.
[43] 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.
[44] 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.
[45] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 174 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360600
[46] 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.
[47] 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.
[48] 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.
[49] 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).
[50] 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.
[51] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 579–594.
[52] Qian Yang, J Jenny Li, and David M Weiss. 2009. A survey of coverage-based testing tools. *Comput. J.* 52, 5 (2009), 589–597.
[53] 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* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532
[54] 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.
[55] 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.