Coverage-Directed Differential Testing of JVM Implementations

Yuting Chen
Department of Computer Science and Engineering
Shanghai Jiao Tong University, China
chenyt@cs.sjtu.edu.cn

Ting Su
Shanghai Key Laboratory of Trustworthy Computing
East China Normal University, China
tsuletgo@gmail.com

Chengnian Sun Zhendong Su
Department of Computer Science
University of California, Davis, USA
{cnsun, su}@cs.ucdavis.edu

Jianjun Zhao
Department of Computer Science and Engineering
Shanghai Jiao Tong University, China
Department of Advanced Information Technology
Kyushu University, Japan
zhao-jj@cs.sjtu.edu.cn

Abstract

Java virtual machine (JVM) is a core technology, whose reliability is critical. Testing JVM implementations requires painstaking effort in designing test classfiles (`*.class`) along with their test oracles. An alternative is to employ binary fuzzing to differentially test JVMs by blindly mutating seeding classfiles and then executing the resulting mutants on different JVM binaries for revealing inconsistent behaviors. However, this blind approach is not cost effective in practice because most of the mutants are invalid and redundant.

This paper tackles this challenge by introducing *classfuzz*, a *coverage-directed fuzzing approach* that focuses on representative classfiles for differential testing of JVMs' startup processes. Our core insight is to (1) mutate seeding classfiles using a set of predefined mutation operators (mutators) and employ *Markov Chain Monte Carlo (MCMC) sampling* to guide mutator selection, and (2) execute the mutants on a reference JVM implementation and use *coverage uniqueness* as a discipline for accepting representative ones. The accepted classfiles are used as inputs to differentially test different JVM implementations and find defects.

We have implemented classfuzz and conducted an extensive evaluation of it against existing fuzz testing algorithms. Our evaluation results show that classfuzz can enhance the ratio of discrepancy-triggering classfiles from 1.7% to 11.9%. We have also reported 62 JVM discrepancies, along with the test classfiles, to JVM developers. Many of our reported issues have already been confirmed as JVM defects, and some even match recent clarifications and changes to the Java SE 8 edition of the JVM specification.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Testing tools (e.g., data generators, coverage testing); D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects; D.3.4 [*Programming Languages*]: Processors—Code generation

General Terms Algorithms, Reliability, Languages

Keywords Differential testing, fuzz testing, Java virtual machine, MCMC sampling

1. Introduction

Java Virtual Machine (JVM) is a mature Java technology. A JVM is responsible for loading, linking, and executing Java classfiles (`*.class`) in the same way on any platform [29]. Various JVMs (*i.e.*, JVM implementations), such as Oracle's HotSpot [3], IBM's J9 [4], Jikes RVM [5], Azul's Zulu [6], and GNU's GIJ [2], are available and still evolving. They adopt different implementation techniques (such as just-in-time compilation, ahead-of-time compilation, and interpretation) and adapt to different operating systems and CPU architectures. To ensure their compatibility, they must consistently implement a single JVM specification [25].

The reality is that no two JVMs are exactly alike, and they can behave discrepantly when encountering corner cases or invalid classfiles: a Java class can run on some JVMs but not

PLDI'16, June 13–17, 2016, Santa Barbara, CA, USA
© 2016 ACM. 978-1-4503-4261-2/16/06...$15.00
http://dx.doi.org/10.1145/2908080.2908095

on the others, applications can be vulnerable when running on some JVMs, or JVMs themselves can crash at runtime. For instance, Lucene, a text search engine, runs amok of bugs on JVMs from different vendors: it fails on any HotSpot release with the `G1` garbage collector; `readVInt()` returns wrong results when Lucene runs on HotSpot for Java 6 or 7u1; and `FST.pack()` may produce corrupted indices when Lucene runs on J9 because of a miscompiled loop, *etc.*[1]

These JVM discrepancies are mainly created under two circumstances. First, JVM implementations can contain defects. Thus they can behave differently from each other because developers may not make the same mistake(s) when developing their respective JVMs. Here we view any programming bug (*e.g.*, null pointer) in a JVM implementation or any conformance violation of the implementation to the JVM specification as a *JVM defect*. In particular, JVM has a 590-page specification [25] which is comprehensive, but can still be vague and ambiguous. The specification also leaves certain behavior undefined: "*if some constraint (a 'must' or 'must not') . . . is not satisfied at run time, the behavior of the Java Virtual Machine is undefined*" (see §6.1 in the JVM specification). A JVM implementation can also be complex. For example, HotSpot has $250K+$ lines of code. Thus it is difficult to have a JVM implementation that strictly conforms to the JVM specification.

Second, there exist compatibility issues among JVM implementations (see Ch. 13 of the Java Language Specification [17]). There also exist mismatches among Java applications, JVM implementations, Java runtime environments (JREs), and platforms. For example, the Java Native Interface (JNI) allows Java code to invoke, or be invoked by, native applications and libraries written in C, C++, and assembly, while it has been reported that in several cases, HotSpot continues running, while J9 crashes [20, 35].

We now formalize JVM *discrepancies* and *defects* to faciliate the presentation of our work. We denote a JVM execution as $r = jvm(e, c, i)$, where jvm is a JVM implementation, e the execution-related environment (including JRE libraries jvm depends on and loaded classes and resources), c and i, respectively, a class running on jvm and the input, and r the observable behavior (*i.e.*, output or errors) reported by jvm.

Definition 1 (JVM Discrepancy)**.** *Given* $r_1 = jvm_1(e_1, c, i)$ *and* $r_2 = jvm_2(e_2, c, i)$*, a* JVM discrepancy *occurs when* r_1 *and* r_2 *are* observably different*, denoted* $r_1 \nsim r_2$*,* i.e.*,* r_1 *and* r_2 *have diverging output or errors.*

A discrepancy can be easily observed if, given the same class, two JVMs behave differently, *e.g.*, one states that the class can run normally, while the other issues an error. When rejecting a classfile, JVMs can give different errors, whose (in)equivalence may require a careful analysis. In our study, we perform a fine-grained analysis of the execution results (details in Section 2.3).

[1] `https://wiki.apache.org/lucene-java/JavaBugs`

Definition 2 (JVM Defect)**.** *We call a JVM discrepancy (*i.e.*, given* $r_1 = jvm_1(e_1, c, i)$ *and* $r_2 = jvm_2(e_2, c, i)$*,* $r_1 \nsim r_2$*) a* JVM defect *when* jvm_1 *and* jvm_2 *operate* w.r.t. *the same environment (*i.e.*,* $e_1 = e_2$*).*

Differential testing [18, 27] can be a promising approach to revealing defects in JVMs by comparing multiple systems. It operates by running each test classfile on the JVMs, and any detected discrepancies (*e.g.*, one classfile can successfully run on one JVM, but not on the others) may indicate defects in JVM implementations. However, differential JVM testing faces two key challenges:

Challenge 1: *Classfiles need to be specifically constructed for differential JVM testing.* Since a classfile can encompass intricate syntactic and semantic constraints, testing engineers have to spend painstaking effort in designing the tests along with their test oracles [24]. Although real-world applications (*e.g.*, Lucene) can be used for testing JVMs, they are insufficient for testing all aspects of JVMs. Real-world applications also introduce difficulties in reproducing, reporting, diagnosing and troubleshooting JVM defects due to their complicated dependencies among applications, JVMs, and JREs [38].

An alternative is to employ binary fuzzing to differentially test JVMs by blindly mutating seed classfiles and running the mutants on different JVM binaries. This approach allows a large number of test classfiles to be produced, including many invalid and unexpected ones. However, it is easy to lead to an enormous number of redundant tests, as most of the mutants trigger a small number of JVM discrepancies.

Challenge 2: *JVM discrepancies are frequent, but mostly only concern compatibility issues, rather than actual defects.* We have conducted a preliminary study by running classfiles in JRE7 libraries on five JVMs, including HotSpot releases for Java 7/8/9, J9 for IBM's SDK8, and GIJ 5.1.0. The results show that 1.7% (364 out of 21736) of the classfiles can reveal JVM discrepancies.[2] For example, the class `sun.beans.editors.EnumEditor` can trigger a `VerifyError` on HotSpot for Java 8 because its superclass `com.sun.beans.editors.EnumEditor` is declared "`final`", thus non-subclassable in JRE8; 37 classfiles can trigger verification errors ("`stack shape inconsistent`") when running on J9 because HotSpot and J9 adopt different stack frames;[3] the JRE7 library classfiles can also trigger many `NoClassDefFoundErrors` and `MissingResourceExceptions` when running on HotSpot releases for Java 8/9, J9, and GIJ, as some classes and resources cannot be loaded by these JVMs at runtime.

These aforementioned discrepancies are mostly related to compatibility problems, rather than JVM defects; they can be eliminated by enforcing JVMs against the specific

[2] `http://stap.sjtu.edu.cn/~chenyt/DTJVM/testingresults.htm`

[3] `http://javarevisited.blogspot.com/2013/01/difference-between-sun-oracle-jvm-ibm.html`

environments (through either upgrading/degrading the JRE libraries or resetting the CLASSPATH environment variables for the JVMs), rather than fixing the JVMs themselves.

To address these challenges, we propose *classfuzz*, a *coverage-directed fuzzing approach* that focuses on constructing *representative* classfiles for differential testing of JVMs' startup processes. A JVM startup process involves the steps of loading, linking, initializing, and invoking classes (Section 2.1 explains it in detail). By representative, we mean that the constructed test classfiles are likely to be different and unique for differential testing, *e.g.*, they take different control-flow paths, enforce new (or new combinations of) checking policies, or lead to new JVM errors/exceptions. For this purpose, we execute the test classfiles on a reference JVM implementation and use *coverage uniqueness* as a discipline for accepting representative ones.

The essential idea of classfuzz is to iteratively mutate seeding classfiles and only accept the representative mutants. The accepted classfiles are used as inputs to differentially test JVMs and find defects. In particular, we define 129 mutators (mutation operators) and employ *Markov Chain Monte Carlo (MCMC) sampling* to guide mutator selection. MCMC sampling is a class of algorithms for sampling from a probability distribution by constructing a Markov chain that converges to the desired distribution [13, 28]. It provides a general solution to many intractable problems without exact algorithms [33].

This paper makes the following main contributions:

1. *Representativeness of tests*. We cast the difficult problem of finding representative tests as a coverage uniqueness problem. This allows us to select those tests confirmed to be representative by running them on a reference JVM. This high-level view is general and applicable in other settings with large-scale software systems and random test inputs.
2. *Coverage-directed classfile mutation*. Classfuzz provides a practical solution to constructing representative test classfiles. As Figure 2 shows, classfuzz (1) mutates the seeding classfiles and creates as many mutants as possible, including many corner cases, (2) runs the mutants on a reference JVM implementation and uses coverage uniqueness to select representative ones, and (3) leverages MCMC sampling to guide mutator selection dynamically as certain mutators are more effective in creating representative classfiles.
3. *Implementation and evaluation*. We have implemented classfuzz and evaluated it extensively against existing fuzz testing algorithms. Our results show that classfuzz can enhance the ratio of discrepancy-triggering classfiles from 1.7% to 11.9%. We have also reported a number of JVM discrepancies, along with the test classfiles, to JVM developers. These discrepancies are mostly defect-indicative or corresponding to the respective checking and verification policies taken by the JVMs, and many of our reported issues have already been confirmed as JVM defects. Some discrepancies even match recent clarifications and changes to the Java SE 8 edition of the JVM specification. We believe that classfuzz is practical and effective in detecting JVM defects and improving the robustness of JVM implementations.

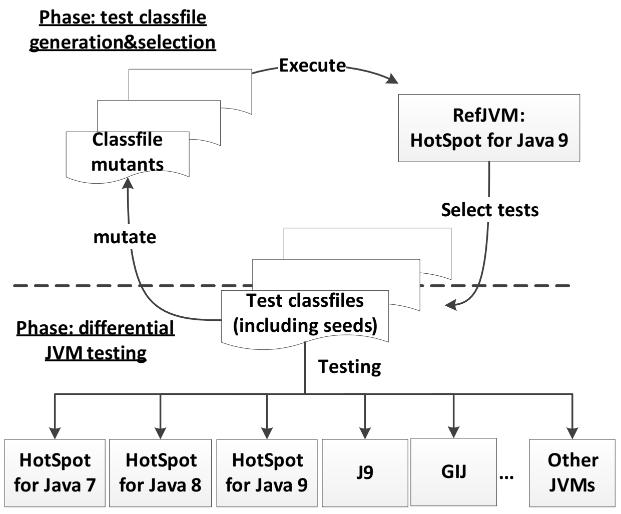

Phase: test classfile generation&selection
Execute
RefJVM: HotSpot for Java 9
Classfile mutants
mutate
Select tests
Test classfiles (including seeds)
Phase: differential JVM testing
Testing
HotSpot for Java 7
HotSpot for Java 8
HotSpot for Java 9
J9
GIJ
...
Other JVMs

Figure 1: An overview of the classfuzz approach.

The rest of this paper is structured as follows. Section 2 presents the technical details of classfuzz. Section 3 describes our extensive evaluation of classfuzz. Section 4 surveys related work, and Section 5 concludes.

2. Approach

This section discusses the necessary background on JVM testing, and introduces our coverage-directed approach to exploiting representative test classfiles.

2.1 Background: Class Format and JVM Startup

A classfile is structurally organized, containing the compiled code for a Java class or interface. It is composed of a magic number ('0xCAFEBABE'), an MD5 checksum value, a constant_pool table, and several executable bytecode segments. Figure 2 shows a simplified classfile M1436188543.class. The major and the minor versions indicate with which platform the classfile is compatible. For example, the major version 51 denotes that the class is compatible with the J2SE 7 platform. The constant pool stores the necessary symbolic information. The class body contains a static initialization method <clinit> (*i.e.*, public abstract {};) and another main method with a code segment. The flags are used to represent the attributes of the class (and its methods and fields) and their access permissions. For example, both methods have the ACC_PUBLIC flag, indicating that the methods can be accessed from outside its package; <clinit> has the ACC_ABSTRACT flag, indicating that the method is abstract and must not be instan-

Table 1: A JVM startup process and types of errors/exceptions that can be thrown

Phase	Possible errors and exceptions
Creation&Loading: class loader tries to find a classfile, say `MyClass`	ClassCircularityError, ClassFormatError, NoClassDefFoundError, *etc.*
Linking: JVM links `MyClass`, and performs verification, preparation, and (optionally) resolution	VerifyError, IncompatibleClassChangeError (AbstractMethodError/IllegalAccessError/InstantiationError/NoSuchFieldError /NoSuchMethodError), UnsatisfiedLinkError, *etc.*
Initializing: JVM executes any class variable initializers, static initializers of `MyClass` and the other linked classes	NoClassDefFoundError, ExceptionInInitializerError, *etc.*
Invoking&Execution: JVM invokes the `main` method of `MyClass`. Notice that executing the JVM instructions constituting the `main` method may cause linking (and consequently creating) additional classes and interfaces, as well as invoking additional methods	Main method is not found, runtime errors/exceptions, user-defined errors/exceptions, *etc.*

```
...
 MD5 checksum 8
     fb69050bbcb9a83ddd90ae393368c5e
...
class M1436188543
 minor version: 0
 major version: 51
 flags: ACC_SUPER
Constant pool:
 ...
  #7 = Utf8 <clinit>
  #8 = Utf8 ()V
  #9 = Class #19 // java/lang/System
  #10 = Utf8 Code
  #11 = Utf8 main
 ...
{
 public abstract {};
  flags: ACC_PUBLIC, ACC_ABSTRACT
 public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
 Code:
  stack=2, locals=1, args_size=1
  0: getstatic #12 // Field java/lang/System.
       out:Ljava/io/PrintStream;
  3: ldc #4 // String Completed!
  5: invokevirtual #21 // Method java/io/
       PrintStream.println:(Ljava/lang/String
       ;)V
  8: return}
```

Figure 2: An example of a decompiled classfile. The `<clinit>` method is incorrectly defined and can trigger a JVM discrepancy: HotSpot can invoke the class normally, while J9 reports a format error "`no Code attribute specified...method=<clinit>()V, pc=0`". This example corresponds to some recent changes to the JVM specification (see Section 3.3).

tiated; the `main` method is typically static and thus has the `ACC_STATIC` flag, *etc.*. The code segment is composed of the opcode specifying the operations to be performed.

A JVM starts up by loading a class and invoking its `main` method [17]. As Table 1 shows, a JVM startup process involves the steps of loading, linking, initializing, and invoking class(es). At any step, the JVM may throw built-in errors and/or exceptions when any constraints are violated. For example, when running on J9, `M1436188543.class` triggers a format error that `<clinit>` lacks a code attribute.

2.2 Coverage-Directed Classfile Mutation

Classfuzz exploits representative classfiles to differentially test JVMs. As Algorithm 1 shows, classfuzz selects an initial corpus of seeds and iteratively mutates them (Section 2.2.1). Let each classfile be executed on a reference JVM. Any mutant, if accepted as a test for differential testing, must take a unique execution path when running on the reference JVM, *i.e.*, any two tests within the test suite should take different execution paths (Section 2.2.3). New, representative classfiles are further taken as seeds (lines 5 and 14 in Algorithm 1) because as we will show in Section 3.2, it is easier to create representative classfiles through mutating representative seeds than mutating non-representative ones.

2.2.1 Mutating Classfiles

Classfuzz can generate a new classfile by randomly selecting a seed and mutating it. We utilize Soot [39] to mutate the seeds because it is a state-of-the-art Java analysis and transformation framework and provides rich APIs for parsing, transforming, and rewriting classfiles. We also define 129 mutators, allowing the seeds to be mutated via rewriting their syntactic structures and intermediate representations; illegal constructs can happen to be introduced into test classfiles, which will further raise JVM discrepancies if they are handled by JVMs differently.[4]

Table 2 shows some typical mutators. 123 out of the 129 mutators are designed for mutating classfiles at the syntactic level. Classfuzz reads a classfile as a SootClass (*i.e.*, the internal structure of the classfile), and then uses a mutator to rewrite it (*e.g.*, modify its modifiers, change its superclass, add an interface, rename a field or method, or delete an exception thrown). The resulting SootClass is then dumped to a new classfile.

[4] A complete list of the 129 mutators can be found at `http://stap.sjtu.edu.cn/~chenyt/DTJVM/classfuzz.htm#_Mutator`.

Table 2: Typical mutators.

What to mutate	Typical mutators	Example (Jimple code before mutation → Jimple code after mutation)
Class	Reset its attributes (*e.g.*, modifiers, name, package name, superclass), *etc.*	`class M1437185190 extends java.lang.Object` → `private class M1437185190 extends java.lang.Thread`
Interface	Insert one or more class-implementing interfaces, delete one or more interfaces, *etc.*	`class M1437185190 extends java.lang.Object` → `class M1437185190 extends java.lang.Object implements java.security.PrivilegedAction`
Field	Insert one or more class fields, delete one or more fields, choose a field and set its attribute(s) (*e.g.*, name, modifiers), *etc.*	`...protected final java.util.Map MAP; ...` → `...protected final java.util.Map MAP; public java.lang.Object MAP; ...`
Method	Insert one or more class methods, delete one or more methods, choose a method and set its attribute(s) (*e.g.*, name, modifiers, return type), *etc.*	`public void <init>(){...}` → `public static void <init>(){...}`
Exception	Select a method and insert one or more exceptions thrown, delete one or more exceptions thrown, *etc.*	`public static void main(java.lang.String[]){` → `public static void main(java.lang.String[]) throws sun.java2d.pisces.PiscesRenderingEngine$2 {`
Parameter	Select a method and insert one or more parameters, delete one or more parameters, *etc.*	`public static void main(java.lang.String[])` → `public static void main(java.lang.Object, java.lang.String[])`
Local variable	Select a method and insert one or more local variables, delete one or more local variables, choose a local variable and set its attribute(s) (*e.g.*, type, name, modifiers), *etc.*	`public static void main(java.lang.String[]) {int $i0; ...}` → `public static void main(java.lang.String[]) { java.lang.String $i0; ...}`
Jimple file	Insert one or more program statements, delete one or more program statements, *etc.*	`r0:=parameter0: java.lang.String[]; r1:=<java.lang.System: java.io.PrintStream out>; virtualinvoke $r1.<java.io.PrintStream:void println(java.lang.String)>("Executed");` → `r0:=parameter0: java.lang.String[]; virtualinvoke $r1.<java.io.PrintStream:void println(java.lang.String)>("Executed"); r1:=<java.lang.System: java.io.PrintStream out>;`

Six mutators are designed for mutating classfiles through rewriting their Jimple files (*i.e.*, the intermediate representations of the classes [7, 39]). Given a classfile, classfuzz transforms it to a Jimple file. It then inserts, deletes, or replaces one or more lines of the Jimple file, which may stochastically change the control flow and/or the syntactic structure of the class. After mutation, classfuzz transforms the mutated Jimple file to another classfile.

To facilitate further JVM testing and result analysis, we supplement each classfile mutant with a simple `main` method that simply prints a message that the class can be normally loaded and the `main` method be normally executed on a JVM. Thus when running on a JVM, a mutated classfile can either be normally invoked or trigger an error/exception when the JVM fails in loading and invoking this mutant.

2.2.2 Mutator Selection

The 129 mutators are not equally effective. In our study, classfuzz rarely creates representative classfiles when using certain mutators. This observation underlies our design to selectively apply mutators for generating representative classfiles.

Proposition: The more representative classfiles have been created by a mutator, the more likely the mutator should be selected for further mutations.

Classfuzz adopts the Metropolis-Hastings algorithm [13, 28] for guiding mutator selection. The Metropolis-Hastings algorithm is an MCMC method for obtaining random samples from a probability distribution. It works by generating a sequence of samples whose distribution closely approximates the desired distribution. Samples are produced iteratively, where the acceptance of the next sample (say s_2) depends only on the current one (say s_1). In our setting, samples correspond to mutators to be selected.

Let the *geometric distribution* be the desired distribution that meets the proposition. A geometric distribution is the probability distribution of the number X of Bernoulli trials needed to obtain one success: If the probability of success on each trial is p, the probability that the k^{th} trial is the first success is given by

$$\Pr(X = k) = (1 - p)^{k-1} p$$

for $k = 1, 2, 3, \ldots$. Here we have

Algorithm 1 Coverage-directed algorithm to generate representative test classfiles

```
Input: $Seeds$: a corpus of seeding classfiles; $mutators$: an array of mutators $[mu_1, \ldots, mu_{129}]$; $TimeBudget$: the time budget
Output: $TestClasses$: a set of test classfiles; $GenClasses$: a set storing all of the generated classfiles
1: $TestClasses \leftarrow Seeds$
2: $GenClasses \leftarrow \emptyset$
3: $mu_1 \leftarrow Random.choice(mutators)$
4: repeat
5:     $cl_i \leftarrow Random.choice(TestClasses)$
6:     $k_1 \leftarrow mutators.index(mu_1)$
7:     repeat
8:         $mu_2 \leftarrow Random.choice(mutators)$
9:         $k_2 \leftarrow mutators.index(mu_2)$
10:    until $(Random.random() >= (1-p)^{k_2-k_1})$
11:    $cl'_i \leftarrow mutate(mu_2, cl_i)$
12:    $GenClasses \leftarrow GenClasses \cup \{cl'_i\}$
13:    if $cl'_i$ is representative w.r.t. $TestClasses$ then
14:        $TestClasses \leftarrow TestClasses \cup \{cl'_i\}$
15:    update the success rate of $mu_2$
16:    $mutators \leftarrow sort(mutators)$ /*sort mutators in descending order of their success rates*/
17:    $mu_1 \leftarrow mu_2$
18: until the time budget is used up
19: $TestClasses \leftarrow TestClasses \setminus Seeds$
20: return $TestClasses$, $GenClasses$
```

$$\Pr(X = 1) = p.$$

We associate with each mutator, mu, a success rate

$$succ(mu) = \frac{\text{\#representative classfiles created by } mu}{\text{\#times that } mu \text{ has been selected}}$$

which computes how frequently mu has been observed to create representative classfiles. The mutators can then be sorted in $mutators$, an array storing mutators in descending order of their success rates: the first mutator in $mutators$ has the highest success rate, while the last the lowest.

The MCMC process allows the sampled mutators to meet the geometric distribution such that the mutators with high success rates are more easily selected than those with low success rates. The Metropolis-Hasting algorithm (*i.e.*, lines 6-10 in Algorithm 1) selects mutators at random, and accepts or rejects the mutators by a Metropolis choice

$$A(mu_1 \to mu_2) = \min\left(1, \frac{\Pr(mu_2)}{\Pr(mu_1)} \frac{g(mu_2 \to mu_1)}{g(mu_1 \to mu_2)}\right)$$

where $g(mu_1 \to mu_2)$ is the proposal distribution describing the conditional probability of proposing a new sample mu_2 given mu_1.

Since these mutators are selected at random, the proposal distribution in our setting is symmetric. The acceptance probability is thus reduced to

$$\begin{aligned} A(mu_1 \to mu_2) &= min(1, \frac{\Pr(mu_2)}{\Pr(mu_1)}) \\ &= min(1, (1-p)^{k_2-k_1}) \end{aligned}$$

where k_1 and k_2 are the indices of mu_1 and mu_2 in $mutators$, respectively. The importance of $A(mu_1 \to mu_2)$ is: if mu_2 has a higher success rate than mu_1, mu_2 is always accepted; otherwise the proposal is selected with a certain probability $(1-p)^{k_2-k_1}$. Furthermore, the bigger $k_2 - k_1$ is, the less the acceptance probability is. Notice that at each iteration the success rates need to be re-calculated and $mutators$ be re-sorted (lines 15-16 in Algorithm 1).

Parameter estimation Let ε be a very small deviation (*e.g.*, 0.001). We estimate the parameter p by requiring it to satisfy three conditions:

$$\begin{aligned} 1 &\geq \sum_{k=1}^{129} \Pr(X = k) \geq 0.95; \\ p &> \frac{1}{129}; \\ \varepsilon &< (1-p)^{129-1} p. \end{aligned}$$

The first condition guarantees that the accumulative probability approaches 1. The second condition ensures that the probability for selecting the first mutator (with the highest success rate) must be larger than $\frac{1}{129}$. The third condition ensures that the mutator with the lowest success rate still has some chance to be selected. Thus the initial value of p needs to be in the range $(0.022, 0.025)$. In our study, we let p be $\frac{3}{129}$ (≈ 0.023).

2.2.3 Accepting Representative Classfiles

In order to evaluate the mutated tests, Tsankov, Dashti, and Basin have defined semi-valid coverage, a coverage criterion which computes how many semi-valid inputs (*i.e.*, inputs that satisfy all correctness constraints but one) are used during a testing session [37]. However, the semi-valid coverage criterion does not fit our setting as it becomes impossible to derive all the constraints from the informal JVM specification; neither does there exist any solution to creating semi-valid classfiles for JVM testing.

Classfuzz adopts a simple strategy by executing the mutated classfiles on a reference JVM *RefJVM* (*e.g.*, HotSpot) and only accepting the representative ones as tests (for further differential testing uses). In particular, we use *coverage uniqueness* as a discipline to accept or reject classfiles.

Let *TestClasses* be a set of classfiles and *cl* (*cl* $\notin$ *TestClasses*) a candidate classfile. Let tr_{cl} be an execution tracefile recording which code statements and branches of *RefJVM* are hit by *cl* when it runs on *RefJVM*. The

Table 3: JVMs used in differential testing.

JVM implementation	Java version supported	Description	Release date
HotSpot for Java 9 [3]	1.9.0-internal	Open-source reference implementation of the JVM specification, Java SE 9 Edition	Late 2016
HotSpot for Java 8 [3]	1.8.0	Open-source reference implementation of the JVM specification, Java SE 8 Edition	March 2014
HotSpot for Java 7 [3]	1.7.0	Open-source reference implementation of the JVM specification, Java SE 7 Edition	July 2011
J9 for IBM SDK8 [4]	1.8.0	IBM's fully compatible JVM with the libraries of Oracle's Java SE 8 platform	February 2015
GIJ 5.1.0 [2]	1.5.0	The GNU Interpreter for Java (GIJ), a Java bytecode interpreter	April 2015

classfile cl is representative *w.r.t.* $TestClasses$ if its tracefile is different from those of the classfiles within $TestClasses$.

We can further alleviate the problem of tracefile comparison by directly comparing their coverage statistics. Let $tr_{cl}.stmt$ and $tr_{cl}.br$ be the statement and branch coverages of cl, respectively. We define the next three criteria that allow the coverage uniqueness to be conveniently checked:

1. $[st]$: $\nexists t \in TestClasses(tr_{cl}.stmt = tr_t.stmt)$;
2. $[stbr]$: $\nexists t \in TestClasses(tr_{cl}.stmt = tr_t.stmt \wedge tr_{cl}.br = tr_t.br)$;
3. $[tr]$: $\nexists t \in TestClasses(tr_{cl}.stmt = tr_t.stmt = (tr_{cl} \oplus tr_t).stmt \wedge tr_{cl}.br = tr_t.br = (tr_{cl} \oplus tr_t).br)$.

$[st]$ requires cl to be different from the classfiles in $TestClasses$ in terms of its statement coverage, and $[stbr]$ in terms of both of its statement and its branch coverage metrics. $[tr]$ requires the tracefile of cl to be statically different from that of any classfile in $TestClasses$ (the execution order of program statements and branches and their frequencies are omitted). Here $\oplus$ is an operator for merging two tracefiles; the formula $tr_{cl}.stmt = tr_t.stmt = (tr_{cl} \oplus tr_t).stmt$ denotes that cl and t cover the same set of program statements of $RefJVM$; $tr_{cl}.br = tr_t.br = (tr_{cl} \oplus tr_t).br$ denotes that the two classfiles cover the same set of branches. Compared with the cost of using $[stbr]$, $[tr]$ incurs extra cost of merging tracefiles.

2.3 Differential Testing of JVMs

In our work, we execute the classfiles on five state-of-the-art JVM implementations shown in Table 3, and compare the execution results for revealing inconsistent behaviors.

All of the JVMs can be started up by executing the command: `java classname`. Let each test result be simplified, by checking whether the `main` method can be normally invoked or during which phase an error/exception is thrown, to (0) "normally invoked", (1) "rejected during the loading phase", (2) "rejected during the linking phase", (3) "rejected during the initialization phase", or (4) "rejected at runtime". The five test outputs *w.r.t.* a classfile can then be encoded into a sequence of bits, and a discrepancy appears when the sequence is not of the same bits (*e.g.*, all zeros and all ones). Figure 3 illustrates an encoded sequence indicating that a classfile is normally invoked by the three HotSpot releases, but rejected by J9 and GIJ in different phases. Notice that the

	HotSpot for Java 7	HotSpot for Java 8	HotSpot for Java 9	J9	GIJ
Outputs	0	0	0	1	2

Figure 3: An encoded sequence of test outputs. Theoretically, there are 5^5 possibilities.

simplification can raise both false positives and negatives in practice because the JVMs may report different errors when one classfile contains multiple illegal constructs, while it is possible that these errors are thrown during the same phase.

Meanwhile, when a discrepancy appears, there do not exist any models or automated means for identifying the root cause of the discrepancy, although many general purposed test case reducers (*e.g.*, AST-Based reducer and the modular reducer [31] for CSmith [41]) do exist for pruning the large irrelevant portions from test inputs. For our purpose, the hierarchical delta debugging algorithm [30] is adapted to reduce a discrepancy-triggering classfile in two steps:

Step 1. Given a classfile cl which triggers a discrepancy (say o), delete one of its methods, statements, fields from its Jimple code and re-generate the classfile (say cl');

Step 2. Retest cl' on the five JVMs. If o retains, we assign cl' to cl; otherwise cl' is discarded.

An engineer can repeat this process until a sufficiently simple classfile that can still trigger o is obtained. With a simplified classfile and the corresponding test outputs, it becomes relatively easy to determine which class component(s) and/or attribute(s) lead to that discrepancy. It also becomes easy to determine whether the discrepancy is caused by compatibility problems or by JVM defects.

3. Evaluation

We have implemented classfuzz and conducted an extensive evaluation of it against existing fuzz testing algorithms. Our evaluation results show that 11% of the representative classfiles can trigger JVM discrepancies, and MCMC sampling helps produce an additional 43% of representative classfiles. We have also reported 62 JVM discrepancies, along with the test classfiles, to JVM developers. These discrepancies are mostly defect-indicative, or corresponding to the respective checking and verification policies taken by the JVMs. Some discrepancies even match recent clarifications and changes to the Java SE 8 edition of the JVM specification. The rest of this section presents our detailed results and analysis.

3.1 Setup

Our empirical evaluation is designed to answer the following research questions:

- *Classfiles*: How many test classfiles can be generated by classfuzz within a limited time period?
- *Effectiveness*: How effective are the test classfiles in a classfuzz-generated test suite?
- *Defects*: Can the test classfiles find any JVM defects?

3.1.1 Preparation

Our classfile mutation and differential testing were conducted on a 64-bit Ubuntu 14.04 LTS desktop (with an Intel Core i7-4770 CPU and 16GB RAM). As Figure 2 shows, we chose HotSpot for Java 9 as a reference JVM because (1) as a forthcoming JVM implementation, this release is supposed to provide the richest functionalities among all of the existing JVM implementations, and (2) this release is open-sourced and allows code coverage to be conveniently collected (using the `--enable-native-coverage` flag). We used the mature, widely adopted coverage tool GCOV + LCOV to collect coverage statistics. Since the reference implementation has 260K lines of code (LOCs), it takes 30+ minutes to collect coverage statistics for each run. Thus we chose `/hotspot/src/share/vm/classfile/`, a package containing the shared files used in a JVM startup process, for coverage analysis. The package contains $11,977$ LOCs and $9,197$ program branches. The cost for coverage analysis is reduced to about 90 seconds per run.

As for testing, our test classfiles were executed on the five JVMs shown in Table 3. All the JVMs except HotSpot for Java 9 are widely used in practice.

We randomly selected $1,216$ classfiles from the JRE7 libraries as the seeds. We did not use all of JRE7's classfiles because, given a limited time period, a smaller number of seeds can be more sufficiently mutated.

Furthermore, we assigned each mutant a major version 51 because a version 51 class is compatible with J2SE7 and thus can be mutated by Soot and recognized by all of the five JVMs. Notice that a JVM may use different algorithms for verifying classfiles of different versions. Thus, it is possible that HotSpot accepts some dubious/illegal constructs in a version 46 class but rejects them if they appear in a version 51 class. How to create classfiles with different versions for revealing JVM defects is beyond the scope of this paper.

3.1.2 Evaluated Methodology

Classfuzz can utilize one of the three uniqueness criteria (*i.e.*, $[st]$, $[stbr]$, and $[tr]$). In order to investigate which criterion is well matched with the approach in practice, we let classfuzz adopt these criteria, respectively. Correspondingly, the algorithms are marked as *classfuzz*$[st]$, *classfuzz*$[stbr]$, and *classfuzz*$[tr]$.

No open-source fuzz testing tool can be directly compared with classfuzz. AFL fuzzer is a security-oriented fuzzer that employs compile-time instrumentation and genetic algorithms to discover test inputs for triggering new internal states in the targeted binary [1]. However, test classfiles cannot be successfully generated by AFL fuzzer due to many simple constraints, *e.g.*, any mutation of a classfile will make the file violate its checksum value, the seeding classfiles are too large to change, *etc.*. Thereafter, we evaluated classfuzz against three other algorithms that we designed for comparison to demonstrate classfuzz's capability:

- *Randfuzz*: It mutates the seeding classfiles at random;
- *Greedyfuzz*: It is a greedy mutation algorithm. Greedyfuzz measures the accumulative coverage of the test suite, and accepts a mutant only when it leads to increased code coverage; and
- *Uniquefuzz*: Similar to classfuzz, uniquefuzz accepts a mutant only when it is unique *w.r.t.* the test suite. It differs from classfuzz in that it selects the mutators randomly. In the evaluation, uniquefuzz only takes $[stbr]$ as its uniqueness criterion.

All algorithms but randfuzz are coverage-directed to accept/reject mutants, requiring code coverage to be computed at each iteration. Each algorithm takes the initial set of seeds, but follows its respective strategy to mutate the seeds and mutants. All algorithms employ the 129 mutators in Section 2.2.1 for classfile mutations. To facilitate discussion, we mark the sets of seeds, classfiles generated, and classfiles selected for JVM testing as $Seeds$, $GenClasses$, and $TestClasses$, respectively, and use $GenClasses_X$ and $TestClasses_X$ to denote the two sets produced by algorithm X.

3.1.3 Metrics

We recorded several metrics during the evaluation. We ran each algorithm for three days, and then reported the number of iterations ($\#Iterations$) and the size of $GenClasses$ and $TestClasses$. To account for randomness in the algorithms, we executed each algorithm five times, but only chose one test suite (*i.e.*, $TestClasses$) with the largest size among the five resulting test suites and the corresponding $GenClasses$.

We computed the success rate of an algorithm X as

$$succ(X) = \frac{|TestClasses_X|}{\#Iterations} \times 100\%$$

which shows the capability of the algorithm in generating test classfiles. We also reported the time spent on generating each classfile.

We evaluated the effectiveness of a test suite through counting the discrepancies it has revealed when running on the five JVMs. Thus we ran each set of classfiles (say

$Classes$) on the targeted JVMs and computed its difference rate as

$$diff = \frac{|Discrepancies|}{|Classes|} \times 100\%$$

where $Discrepancies$ only includes the classfiles that can trigger JVM discrepancies. $diff$ can be used to estimate the inconsistencies among the JVMs: since each representative classfile corresponds to a unique test scenario, the higher $diff$ is, the more inconsistent the five JVMs are in processing classfiles.

With the encoded test results, we evaluated the effectiveness of each test suite in revealing various discrepancies. Let two discrepancies be classified into one category when they have the same encoded outputs. We counted the number of *distinct discrepancies* ($|Distinct\_Discrepancies|$) revealed by each test suite. The more distinct discrepancies are found by a test suite, the more types of discrepancies are supposed to be revealed by the suite.

We investigated each class in $TestClasses_{classfuzz[stbr]}$ manually, and reported to JVM developers some discrepancies that are more likely caused by JVM defects.

3.2 Results on Classfile Generation

Table 4 shows the sizes of the test suites. From the results, we observe:

- Finding 1: Randfuzz generates 20 times as many classfiles as those generated by any other coverage-directed algorithm; classfuzz[$stbr$] generates the most number of representative classfiles and achieves the highest success rate among all the coverage-directed algorithms.

The sizes of the test suites can vary. In three days, randfuzz generated 29,523 non-representative classfiles (on average 6.8 classes per minute). Meanwhile, as a human engineer may spend minutes or hours on analyzing one classfile and the test outputs, randfuzz simply shifts the cost of classfile generation to manual result analysis. In contrast, all the directed algorithms produce classfiles more slowly. In three days, they generate $1,432 \sim 1,543$ classfiles (on average 0.35 classes per minute), as time is needed to (1) collect coverage statistics at runtime; and (2) check each candidate classfile's uniqueness *w.r.t.* the test suite. Classfuzz[tr] also incurs extra cost in merging tracefiles. Notice that the cost of mutator selection is quite low and can be omitted in practice.

The success rates of the algorithms range from 5.1% to 63.7%. Classfiles are not generated during some iterations because (1) some seeds are invalid and thus cannot be used as inputs for mutation, or (2) some SootClasses or Jimple files after rewritten are invalid (*e.g.*, the constant pool may be incomplete) and their classfiles cannot be further processed by Soot. Directed algorithms (*i.e.*, classfuzz, greedyfuzz, and uniquefuzz) also discard $41.7 \sim 93.2\%$ of the generated classfiles due to their non-uniqueness.

Classfuzz[$stbr$] and classfuzz[tr] produce more representative tests than classfuzz[st]. One reason is that the two algorithms accept coverage-unique classfiles in a two-dimensional space (*i.e.*, $stmt$ and br), but classfuzz[st] in one-dimensional space. For instance, let two classfiles achieve the code coverage of $4,938/2,604$ and $4,938/2,655$, respectively. Classfuzz[st] takes one classfile as a test, while classfuzz[$stbr$] takes both. Instead, greedyfuzz takes 98 out of $1,432$ classfiles as the representative tests, making differential testing inadequate.

Theoretically, [tr] is stronger than [$stbr$] because tracefiles having the same coverage statistics can still be different. However, in our evaluation, the two criteria are similarly effective in accepting representative classfiles. Specially, the 774 classfiles in $TestClasses_{classfuzz[tr]}$ correspond to 758 unique coverage statistics; only 16 classfiles have the same coverage statistics as the others, but have different tracefiles. We compared the outputs of the 16 classfiles with those of the other classfiles, and observed that for two classfiles having the same coverage statistics, HotSpot for Java 9 always returns the same output.

We randomly selected $1,500$ classfiles from $TestClasses_{randfuzz}$. They correspond to only 237 unique coverage statistics. Comparatively, $GenClasses_{classfuzz[stbr]}$ and $GenClasses_{uniquefuzz}$ correspond to 898 and 628 unique coverage statistics, respectively. It justifies our assumption in Section 2.2, *i.e.*, it can be easier to create representative classfiles by mutating representative seeds than mutating non-representative ones.

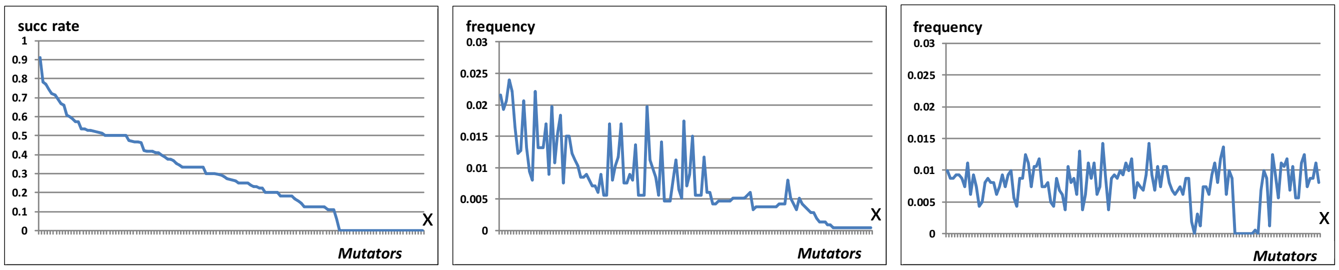
- Finding 2: classfuzz can utilize the prior knowledge to select mutators.

We summarized the mutators selected for generating $TestClasses_{classfuzz[stbr]}$. Figure 4a shows the mutators sorted in descending order of their success rates, and Figure 4b shows their frequencies. Although mutator selection does not strictly meet the geometric distribution because the iterations are insufficient, it clearly agrees with the proposition in Section 2.2.1: the higher the success rate of a mutator, the more frequently the mutator is selected. For example, among all of the mutators, *renaming one method* has been selected for the most number of times (49), and it also achieves a high success rate (75%). In comparison, as Figure 4c shows, uniquefuzz selects the mutators without any guidance, even though some of them rarely create representative classfiles; the frequencies of the mutators vary only because some mutators hardly create classfiles.

Table 5 shows the top ten mutators and their frequencies. These mutators are easy to create a rich set of classfile mutants because they are likely to rewrite program constructs (*e.g.*, methods, initializers, exceptions) that need to be meticulously processed. Some mutators (*e.g.*, those for rewriting the parameter list of a method) are less effective in creating coverage-unique classfiles, because (1) the resulting SootClasses or Jimple files violate some transforming constraints in SOOT and thus cannot be dumped to classfiles, or (2) some

Table 4: Results on classfile generation.

	Classfuzz [stbr]	Classfuzz [st]	Classfuzz [tr]	Uniquefuzz	Greedyfuzz	Randfuzz
$\#iterations$	2130	2108	1971	1898	1911	46318
$\|GenClasses\|$	1539	1543	1450	1436	1432	29523
$\|TestClasses\|$	898	494	774	628	98	29523
$succ$	42.2%	23.4%	39.3%	33.1%	5.1%	63.7%
Average time for each generated class (sec.)	168.4	168.0	178.8	180.5	181.0	8.78
Average time for each test class (sec.)	288.6	524.7	334.9	412.7	2644.9	8.78



(a) Success rates of mutators for generating $TestClasses_{classfuzz[stbr]}$. **(b)** Frequencies of mutators for generating $TestClasses_{classfuzz[stbr]}$. **(c)** Frequencies of mutators for generating $TestClasses_{uniquefuzz}$.

Figure 4: Correlation between the success rates of mutators and their selection frequencies. In either figure, the x-axis denotes a sequence of mutators. For comparison reason, these mutators are sorted in descending order of their success rates for generating $TestClasses_{classfuzz[stbr]}$.

Table 5: Top ten mutators.

What to mutate	Mutator	Succ rate	Frequency
Method	Select a class and replace all of its methods with those of another class	0.913	0.022
Exception	Select a method and add a list of exceptions thrown	0.780	0.019
Class	Select a class and set a specified class as its superclass	0.773	0.021
Method	Select a method and rename it	0.745	0.024
Field	Select a class and replace all of its fields with those of another class	0.723	0.022
Method	Select a method and change its return type	0.714	0.016
Exception	Select a method and add one exception thrown	0.692	0.012
Class	Select a class and set its superclass as a class randomly selected from a class list	0.667	0.013
Local variable	Select a local variable and change its type	0.659	0.021
Method	Select a method and delete it	0.607	0.013

program constructs (or properties) can be simply processed, making the same code be covered for processing the class constructs (or properties) before and after rewritten.

MCMC sampling provides extra benefits in generating representative classfiles. By comparing the test suites of classfuzz$[stbr]$ and uniquefuzz, we can estimate that MCMC sampling helps produce an additional $43\%(= \frac{898-628}{628})$ of representative classfiles within three days.

3.3 Results on Differential JVM Testing

Table 6 summarizes the test results of executing the seeds and the mutated classfiles on the five JVMs. By observing the results, we have two additional findings.

- Finding 3: classfuzz can enhance the ratio of discrepancy-triggering classfiles from 1.7% to 11.9%.

JVM discrepancies can be exaggerated by the classfile mutants: only 364 out of the $21,736$ JRE7 classfiles can trigger JVM discrepancies, while 107 out of the 898 mutants in $TestClasses_{classfuzz[stbr]}$ can. The results suggest that (1) the tested JVM implementations are compatible for most of the classfiles, while still differ in processing some corner cases; and (2) fuzz testing is a promising approach to exposing JVM discrepancies.

Table 7 presents a detailed summary of using the 898 classfile mutants in $TestClasses_{classfuzz[stbr]}$ to differentially test the JVMs: 127, 127, 125, 123, and 145 classfiles are

Table 6: Results on testing of JVMs. Since the results for randfuzz are identical under both cases, we do not repeat them (and labeled as "- ").

	JRE7 classes	seeding classfiles	Classfuzz [stbr]	Classfuzz [st]	Classfuzz [tr]	Uniquefuzz	Greedyfuzz	Randfuzz
$\|GenClasses\|$	21736	1216	1539	1543	1450	1436	1432	–
all invoked	77	17	197	199	226	203	195	
all rejected at the same stage	21295	1162	1100	1104	984	1063	980	
$\|Discrepancies\|$	364	37	242	240	240	170	257	
$\|Distinct\_Discrepancies\|$	16	9	17	14	13	10	15	
$diff$	1.7%	3.0%	15.7%	15.6%	16.5%	11.8%	17.9%	
$\|TestClasses\|$	–	–	898	494	774	628	98	29523
all invoked			121	62	122	109	11	3914
all rejected at the same stage			670	374	564	462	72	20072
$\|Discrepancies\|$			107	58	88	57	15	5537
$\|Distinct\_Discrepancies\|$			17	11	12	9	7	14
$diff$			11.9%	11.7%	11.4%	9.1%	15.3%	18.8%

Table 7: Results on testing of JVMs using the 898 classfile mutants in $TestClasses_{classfuzz[stbr]}$.

	HotSpot for Java 7	HotSpot for Java 8	HotSpot for Java 9	J9	GIJ
Normally invoked	127	127	125	123	145
Rejected during the creation/loading phase	51	51	51	57	34
Rejected during the linking phase	719	719	718	707	714
Rejected during the initialization phase	1	1	4	1	2
Rejected at runtime	0	0	0	0	3

runnable on three HotSpot releases, J9, and GIJ, respectively. J9 rejects the most number of classfiles, while GIJ rejects the least. From the results, we notice that, among the JVMs, GIJ is the most lenient and accepts many more invalid classfiles.

- Finding 4: $TestClasses_{classfuzz[stbr]}$ can reveal the most number of distinct discrepancies among all of the test suites.

Table 6 shows that $TestClasses_{classfuzz[stbr]}$ reveals more distinct discrepancies than the other test suites. The set of distinct discrepancies revealed by $TestClasses_{classfuzz[stbr]}$ can also contain those revealed by the other test suites but $Seeds$. It indicates that $TestClasses_{classfuzz[stbr]}$ is effective in revealing a variety of JVM discrepancies. Instead, randfuzz can produce a significant number of classfiles which trigger $5,537$ JVM discrepancies, while these discrepancies can be reduced to only 14 distinct ones.

$GenClasses$ and $TestClasses$ of classfuzz[$stbr$] can reveal the same number of distinct discrepancies, while the other algorithms cannot. Therefore, from the perspective of distinct discrepancies, $TestClasses_{classfuzz[stbr]}$ can become a replacement of $GenClasses_{classfuzz[stbr]}$ for differentially testing the five JVMs.

Discrepancy analysis JVM discrepancies have not been much studied before. Indeed, they have rarely been reported to JVM developers and were not well-known. We have investigated several bug tracking systems, mailing lists, and forums for Java and JVMs, and selected the most active ones: one bug tracker for Java and two mailing lists for HotSpot.[5] We searched the mailing lists for messages between January, 2013 and December, 2015, and did not find any whose subjects concern JVM discrepancies. We searched the bug tracker for bugs related to GIJ and J9. Only five bugs were retrieved, which were not reported explicitly as JVM discrepancies. It is interesting to note that many changes proposed in these mailing lists and bug trackers may be leveraged to discover JVM discrepancies, which we plan to explore in future work.

Our test classfiles clearly indicate the presences of JVM discrepancies. We manually analyzed the classfiles in $TestClasses_{classfuzz[stbr]}$ and their test outputs, and collected 62 JVM discrepancies along with the simplified classfiles. To the best of our knowledge, these discrepancies were previously unknown: 28 of the 62 discrepancies indicate that defects exist in one or more JVM implementations; 30 discrepancies are caused because JVMs take different verification/checking strategies or hold different accessibilities to resources and libraries; the remaining four correspond to compatibility issues.[6] Some of our reported issues are listed

[5] The bug tracker is `http://bugs.java.com/`, and the two mailing lists are `http://mail.openjdk.java.net/mailman/listinfo/hotspot-dev` and `http://mail.openjdk.java.net/mailman/listinfo/hotspot-runtime-dev`.

[6] The collection can be found at `http://stap.sjtu.edu.cn/~chenyt/DTJVM/classfuzz.htm`.

below. We also explain how classfiles are mutated to discover these problems.

Problem 1: *The clause "other methods named `<clinit>` in a class file are of no consequence" in the JVM specification has caused an unreasonable amount of confusion.*

The program in Figure 2 reveals a behavior discrepancy between HotSpot and J9. In particular, `public abstract <clinit>` can be created either by renaming one abstract method as `<clinit>` or by adding an `ACC_ABSTRACT` flag to the method `<clinit>` and then deleting its opcode. Since the method `<clinit>` does not have the `ACC_STATIC` flag, HotSpot treats it as an ordinary method instead of an initialization method, while J9 throws a `ClassFormatError`.

Our reported issue has been confirmed as a defect in the JVM specification and a bug in J9.[7] It also matches a recent specification clarification.[8] The specification committee will make a revision, in Section 2.9 of the JVM specification, where a class/interface initialization method will be more strictly defined: "*A method is a class or interface initialization method if all of the following are true: (1) It has the special name `<clinit>`; (2) It takes no arguments; (3) It is void; (4) In a class file whose version number is 51.0 or above, the method has its `ACC_STATIC` flag set. Other methods named `<clinit>` in a class file are not class or interface initialization methods*".

A JVM following SE 9 must apply the new definition for all classfile versions. The corresponding Technology Compatibility Kit (JCK) tests are under development.[9]

Problem 2: *JVMs take their own classfile verification and type checking polices.*

Discrepancies can be caused due to the different classfile verification and type-checking polices taken by the JVMs. Methods may become illegal if the statements inside are deleted or new statements are added, and classes with such methods may expose JVMs' discrepancies in format checking and classfile verification. Some discrepancies are unfamiliar to even JVM developers. For example, IBM's VM, J9, is less strict than HotSpot because J9 only verifies a method when it is invoked, while HotSpot verifies all methods before execution.[10] This fact is not well-known.

GIJ can report a verification error for a method in which initialized and uninitialized types are merged, but HotSpot cannot.[11]

HotSpot misses catching some incompatible type castings. For example, the class `M1433982529` is mutated by setting the type of the only parameter of the method `internalTransform` from `java.util.Map` to `java.lang.String`. GIJ throws a verification error for the class because it is unsafe to perform a type casting between `java.lang.String` and `java.util.Map`, while HotSpot does not report any error for this. The same issue occurs in some other type castings, *e.g.*, between `java.lang.Boolean` and `java.util.Enumeration`.

```
//The Jimple code of M1433982529
public class M1433982529 extends java.lang.
    Object{
  protected void internalTransform(java.lang.
      String)  {
   java.util.Map r0;
   r0 := @parameter0: java.util.Map;
   staticinvoke <java.lang.Object: boolean
       getBoolean(java.util.Map)>(r0);
   return;
}}
```

Problem 3: *JVMs are not compatible to access some classes.*

The tested JVMs do not follow the same strategy for accessing certain classes. For example, consider the source code shown below for the class `sun.java2d.pisces.PiscesRenderingEngine`. The class `PiscesRenderingEngine$2` is generated for initializing `NormMode`. We add a thrown exception of type `PiscesRenderingEngine$2` to the `main` method of the class `M1437121261`. HotSpot reports a `java.lang.IllegalAccessError` for the class `M1437121261`, while J9 and GIJ do not.[12]

```
//The source code of sun.java2d.pisces.
    PiscesRenderingEngine
public class PiscesRenderingEngine extends
    RenderingEngine {
  ...
  private static enum NormMode {OFF, ON_NO_AA
      , ON_WITH_AA};
  ...
}
//The Jimple code of M1437121261
public class M1437121261 {
  public static void main (String[] r0)
      throws sun.java2d.pisces.
          PiscesRenderingEngine$2{
    ...
}}
```

Problem 4: *GIJ behaves significantly differently from HotSpot and J9.*

GIJ 5.1.0 conforms to Java 1.5.0, although it can process the version 51 classes. Thus it behaves significantly differently from the other JVMs conforming to Java 1.7.0, 1.8.0, or 1.9.0-internal, while many discrepancies are obvious JVM defects. Some of the differences are:

[7] http://mail.openjdk.java.net/pipermail/jls-jvms-spec-comments/2015-July/000013.html

[8] https://bugs.openjdk.java.net/browse/JDK-8130682

[9] https://bugs.openjdk.java.net/browse/JDK-8135208

[10] http://mail.openjdk.java.net/pipermail/hotspot-runtime-dev/2016-January/017439.html.

[11] http://mail.openjdk.java.net/pipermail/hotspot-runtime-dev/2015-June/015253.html.

[12] One explanation is that the class starts with `sun.*` is not accessible anymore in JDK 9 (with modules), while counterexamples still exist (see http://mail.openjdk.java.net/pipermail/hotspot-runtime-dev/2016-January/017676.html).

- When processing an interface extending a class such as `java.lang.Exception`, either HotSpot or J9 will throw a `ClassFormatError` because the superclass of an interface must be `java.lang.Object`, but GIJ fails in catching this kind of illegal inheritance structures. An interface for triggering this discrepancy can be obtained by changing its superclass.
- All of the JVMs but GIJ require that an interface method must be public and abstract, and an interface field must be public, static, and final. An interface for this can be obtained by changing the modifiers of one of its methods or fields.
- GIJ can execute an interface having a `main` method (*i.e.*, `public static main(String[])`), while the other JVMs cannot. Such an interface can be created by inserting a `main` method into a seeding interface.
- The following method signature `public abstract void <init> (int, int, int, boolean);` is rejected by all JVMs, except GIJ. Hotspot and J9 reject it because `<init>` should not be static, final, synchronized, native or abstract. Similarly, a method signature such as `public java.lang.Thread <init>()` is allowed by GIJ, but forbidden by HotSpot and J9 because `<init>` should not return any result. The classfile mutants for this are created by changing the modifiers or return type of `<init>`.
- GIJ accepts a class with duplicate fields, while the others do not. A classfile mutant for this can be created by inserting one or more class fields that exist in the seed.

4. Related Work

We discuss four strands of related work: (1) directed random testing, (2) mutation-based fuzz testing, (3) JVM testing, and (4) MCMC sampling for testing.

Directed random testing Godefroid, Klarlund, and Sen have introduced DART, a directed random testing approach of intertwined concrete and symbolic execution [15]. DART performs random testing of a program under test, analyzes how the program behaves under random testing, and then automatically generates new test inputs to direct the execution along alternative program paths. Godefroid, Levin, and Molnar have presented another idea of whitebox fuzz testing [16]. The approach records the execution traces of a program and generates constraints capturing how the program uses its inputs. The generated constraints are used to produce new inputs directing the program to follow different control paths.

Despite following the same general idea of searching for different control paths, classfuzz differs from DART and whitebox fuzz testing in that it directs test generation via coverage, rather than symbolic execution and constraint solving. Thus, it is simpler and more generally applicable, while it would be interesting future work to investigate effective combinations of the techniques.

Chen *et al.* have measured the similarities among test inputs and formulated the problem of selecting diverse, interesting test inputs from a large set of random, failure-triggering test cases as a *fuzzer taming problem* [12]. However, taming a fuzzer relies on the quality of the initial test set. The similarities among test classfiles are also not strongly correlated to their strengths in exposing JVM discrepancies. In comparison, classfuzz advocates the idea of generating high-quality test inputs, rather than selecting test inputs.

Mutation-based fuzz testing Fuzz testing is a general approach to generating test inputs for testing complicated software systems. However, as explained in the evaluation section, no open-source fuzz testing tool can be directly compared with classfuzz, because no test classfiles can be directly generated by the state-of-the-art fuzz testing tools.

Domain-specific fuzzers are available for generating test inputs for testing large-scale systems. KEmuFuzzer is a methodology for testing system VMs (such as BOCHS, QEMU, VirtualBox and VMware) on the basis of protocol-specific fuzzing and differential analysis consisting in forcing the VMs to execute specially crafted snippets of code and in comparing their behavior [26].

Fuzz testing has also been employed to find issues related to system calls [14, 40], security protocols [11, 36], language compilers [22] and interpreters [19, 32]. LangFuzz provides a general framework for mutating seeding programs and testing [19]: Test programs are generated by replacing code fragments of a seeding program with those stem from programs which are known to have caused invalid behaviors. Classfuzz differs in that it (1) generates much more "representative"classfiles for differential JVM testing, and (2) selectively applies mutators.

Le *et al.* have presented the Equivalence Modulo Inputs (EMI) approach to validating compilers, where the EMI variants of a program can be generated by profiling the program's executions and stochastically pruning its unexecuted code [22]. EMI may be adapted to produce executable test programs (which is interesting to explore for finding JVM execution discrepancies), but it cannot produce diverse classfiles to test JVM's startup (*i.e.*, initialization/linking/loading).

Domain-specific fuzz testing has also been adapted to application VMs. Sirer and Bershad have advocated an early idea of generating a test classfile by putting a single one-byte value change at a random offset in a base classfile [34]. Kyle *et al.* randomly mutate Dex bytecode in order to reveal defects in Android VMs [21]. These fuzz testing techniques are similar to randfuzz designed in our evaluation part, while the evaluation results have clearly demonstrated that the directed approaches are more effective in revealing JVM discrepancies than the non-directed ones.

JVM testing JVMs need to be tested before their release. Oracle provides an extensive test suite, *i.e.*, the Java Compatibility Kit (JCK), to certify compatible implementations of the Java platform.[13] However, JCK is not aimed at exploiting defects in any released, compatible JVMs.

Sirer and Bershad have proposed lava, a domain specific language for producing a code generator. Running the code-generator on a seeding classfile can produce a number of classfiles [34]. Yoshikawa, Shimura, and Ozawa have designed a generator which generates a classfile by producing its control flow, and then filling the bytecodes into control flow edges [42]. Since a classfile can encompass complicated syntactical and semantical constraints, a generator usually produces simple classfiles. In constrast, classfuzz iteratively mutates seeding classfiles, allowing structurally complicated classfiles to be generated.

Formal models have been developed for JVM testing. Calvagna, Fornaia, and Tramontana model a JVM as a finite state machine for deriving test classfiles [8, 9]. They also define a formal model for the Java Card bytecode syntactical and semantic rules from which test programs can be derived for testing the security of the bytecode verifier [10]. Instead, classfuzz does not rely on any formal model for JVM testing, as it incurs much effort in deriving a precise model from the informal JVM specification.

MCMC sampling for testing Zhou, Okamura, and Dohi have proposed an MCMC Random Testing (MCMCRT) approach which utilizes the prior knowledge and the information on preceding test outcomes for estimating parameters for software testing [43]. However, MCMCRT mainly generates numerical test inputs instead of structured ones.

Le *et al.* have applied MCMC sampling to EMI [22] to explore the search space for finding deep compiler bugs [23], where the program distances are calculated and the MCMC sampler more often draws the test programs with larger program distances. Chen and Su have presented mucert, an approach that employs MCMC sampling to diversify a set of seeding certificates for testing SSL/TLS implementations [11]. Compared with mucert, classfuzz is the first to utilize MCMC sampling to selectively apply mutators and find representative tests.

5. Conclusion

Testing mature JVM implementations and finding JVM defects can be a challenging task. Although domain-aware fuzzing has been employed in exposing defects in large-scale software systems, it has not yet been successfully employed in JVM testing. We have proposed a coverage-directed approach named classfuzz to differential testing of JVMs. Our evaluation results have clearly shown that classfuzz is effective in yielding representative test classfiles and finding JVM defects. We believe that JVM developers can use classfuzz to identify latent flaws in JVM implementations and improve their robustness.

[13] `http://docs.oracle.com/javame/testing/testing.html`

More information on our project on differential JVM testing can be found at `http://stap.sjtu.edu.cn/~chenyt/DTJVM/index.htm`.

Acknowledgements

We would like to thank our shepherd, Martin Rinard, and the anonymous PLDI reviewers for valuable feedback on earlier drafts of this paper. We would also like to thank JVM developers for analyzing our reported issues. This research was sponsored in part by 973 Program in China (Grant No. 2015CB352203), the National Nature Science Foundation of China (Grant No. 61572312, 61272102, 61472242, and 61572313), and United States NSF Grants (Grant No. 1117603, 1319187, 1349528, and 1528133). Yuting Chen was partially supported by Science and Technology Commission of Shanghai Municipality's Innovation Action Plan (No. 15DZ1100305).

References

[1] `https://lcamtuf.coredump.cx/afl/`.

[2] `https://gcc.gnu.org/onlinedocs/gcj/index.html`.

[3] `http://openjdk.java.net`.

[4] `http://www.ibm.com/developerworks/java/jdk/`.

[5] `http://www.jikesrvm.org`.

[6] `http://www.azulsystems.com/products/zulu`.

[7] A. Bartel, J. Klein, Y. L. Traon, and M. Monperrus. Dexpler: converting Android Dalvik bytecode to Jimple for static analysis with Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis (SOAP 2012)*, pages 27–38, 2012.

[8] A. Calvagna and E. Tramontana. Automated conformance testing of Java virtual machines. In *Proceedings of the 7th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS)*, pages 547–552, 2013.

[9] A. Calvagna and E. Tramontana. Combinatorial validation testing of Java Card byte code verifiers. In *Proceedings of the 2013 Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 347–352, 2013.

[10] A. Calvagna, A. Fornaia, and E. Tramontana. Combinatorial interaction testing of a Java Card static verifier. In *Proceedings of the 7th IEEE International Conference on Software Testing, Verification and Validation (ICST 2014)*, pages 84–87, 2014.

[11] Y. Chen and Z. Su. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*, 2015.

[12] Y. Chen, A. Groce, C. Zhang, W. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*, pages 197–208, 2013.

[13] S. Chib and E. Greenberg. Understanding the Metropolis-Hastings algorithm. *The American Statistician*, 49(4):327–335, Nov. 1995.

[14] A. Gauthier, C. Mazin, J. Iguchi-Cartigny, and J. Lanet. Enhancing fuzzing technique for OKL4 syscalls testing. In *Proceedings of the Sixth International Conference on Availability, Reliability and Security (ARES 2011)*, pages 728–733, 2011.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005)*, pages 213–223, 2005.

[16] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008*, 2008.

[17] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 8 Edition*. 2015. URL `http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf`.

[18] A. Groce, G. J. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *Proceedings of the International Conference on Software Engineering (ICSE 2007)*, pages 621–631, 2007.

[19] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 2012)*, 2012.

[20] G. Kondoh and T. Onodera. Finding bugs in Java native interface programs. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 109–118, 2008.

[21] S. C. Kyle, H. Leather, B. Franke, D. Butcher, and S. Monteith. Application of domain-aware binary fuzzing to aid Android virtual machine testing. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2015)*, pages 121–132, 2015.

[22] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI 2014)*, page 25, 2014.

[23] V. Le, C. Sun, and Z. Su. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*, pages 386–399, 2015.

[24] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification: Java SE 7 Edition*. 2013. URL `http://docs.oracle.com/javase/specs/jvms/se7/html/index.html`.

[25] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification: Java SE 8 Edition*. 2015. URL `http://docs.oracle.com/javase/specs/jvms/se8/html/index.html`.

[26] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing system virtual machines. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis (ISSTA 2010)*, pages 171–182, 2010.

[27] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[28] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[29] J. Meyer and T. Downing. *Java Virtual Machine*. O'Reilly, 1997.

[30] G. Misherghi and Z. Su. HDD: hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*, pages 142–151, 2006.

[31] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012)*, pages 335–346, 2012.

[32] J. Ruderman. Introducing jsfunfuzz. URL `http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/`.

[33] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)*, pages 305–316, 2013.

[34] E. G. Sirer and B. N. Bershad. Using production grammars in software testing. In *Proceedings of the Second Conference on Domain-Specific Languages (DSL 1999)*, pages 1–13, 1999.

[35] G. Tan. JNI light: An operational model for the core JNI. In *Proceedings of the 8th Asian Symposium on Programming Languages and Systems (APLAS 2010)*, pages 114–130, 2010.

[36] P. Tsankov, M. T. Dashti, and D. A. Basin. SECFUZZ: fuzz-testing security protocols. In *Proceedings of the 7th International Workshop on Automation of Software Test (AST 2012)*, pages 1–7, 2012.

[37] P. Tsankov, M. T. Dashti, and D. A. Basin. Semi-valid input coverage for fuzz testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2013)*, pages 56–66, 2013.

[38] S. T.V. *Oracle JRockit Diagnostics and Troubleshooting Guide, Release R28*. 2011. URL `http://docs.oracle.com/cd/E15289_01/doc.40/e15059.pdf`.

[39] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research*, page 13, 1999.

[40] V. M. Weaver and D. Jones. perf_fuzzer: Targeted fuzzing of the perf_event_open() system call. Technical Report UMAINE-VMW-TR-PERF-FUZZER, University of Maine, July 2015.

[41] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)*, pages 283–294, 2011.

[42] T. Yoshikawa, K. Shimura, and T. Ozawa. Random program generator for Java JIT compiler test system. In *Proceedings of the 3rd International Conference on Quality Software (QSIC 2003)*, page 20, 2003.

[43] B. Zhou, H. Okamura, and T. Dohi. Markov Chain Monte Carlo random testing. In *Advances in Computer Science and Information Technology*, pages 447–456, 2010.