2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

Deep Differential Testing of JVM Implementations

Yuting Chen* Ting Su† Zhendong Su‡§

*Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
†Nanyang Technological University, Singapore
‡ETH Zurich, Switzerland
§University of California, Davis, USA

Email: *chenyt@sjtu.edu.cn, †tsuletgo@gmail.com, ‡zhendong.su@inf.ethz.ch

***Abstract*—The Java Virtual Machine (JVM) is the cornerstone of the widely-used Java platform. Thus, it is critical to ensure the reliability and robustness of popular JVM implementations. However, little research exists on validating production JVMs. One notable effort is classfuzz, which mutates Java bytecode *syntactically* to stress-test different JVMs. It is shown that classfuzz mainly produces illegal bytecode files and uncovers defects in JVMs' startup processes. It remains a challenge to effectively test JVMs' bytecode verifiers and execution engines to expose deeper bugs.**

This paper tackles this challenge by introducing classming, a novel, effective approach to performing deep, differential JVM testing. The key of classming is a technique, *live bytecode mutation*, to generate, from a seed bytecode file f, *likely* valid, executable (*live*) bytecode files: (1) capture the seed f's *live bytecode*, the sequence of its executed bytecode instructions; (2) repeatedly manipulate the control- and data-flow in f's live bytecode to generate semantically different mutants; and (3) selectively accept the generated mutants to steer the mutation process toward live, diverse mutants. The generated mutants are then employed to differentially test JVMs.

We have evaluated classming on mainstream JVM implementations, including OpenJDK's HotSpot and IBM's J9, by mutating the DaCapo benchmarks. Our results show that classming is very effective in uncovering deep JVM differences. More than 1,800 of the generated classes exposed JVM differences, and more than 30 triggered JVM crashes. We analyzed and reported the JVM runtime differences and crashes, of which 14 have already been confirmed/fixed, including a highly critical security vulnerability in J9 that allowed untrusted code to disable the security manager and elevate its privileges (CVE-2017-1376).

***Keywords*—Differential JVM testing; live bytecode mutation; semantically different mutants**

I. Introduction

The Java platform has been in widespread use, and the Java Virtual Machine (JVM) is its cornerstone to run Java applications safely and portably [1] [2]. Defects in JVM implementations can lead to unexpected behavior or security breaches, since a JVM implementation runs bytecode generated by (Java) compilers, but also any bytecode, including byte-code-engineered variants or even ones from attackers. However, few techniques exist to help systematically validate production JVMs (*e.g.*, HotSpot [3] [4], IBM's J9 [5], Azul's Zing [6] and Zulu [7], and the Jikes RVM [8] [9]) and improve their robustness. One promising approach is to *differentially testing JVMs* — running the same Java bytecode (`*.class`) on different JVMs to expose their differences.

The state-of-the-art technique is classfuzz [10], which mutates Java classes *syntactically* (*e.g.*, by changing its modifiers or the type of a variable) to differentially test JVMs' *startup processes* (*i.e.*, loading, linking, and initialization). Despite its effectiveness in exposing differences in the JVMs' startup processes, classfuzz cannot adequately exercise JVMs' bytecode verifiers and execution engines at the backend.

Figure 1 illustrates the high-level process of how a JVM runs a class — the bytecode verifier ensures that each class satisfies the necessary constraints at link time [2] [11], and subsequently the execution engine is responsible for just-in-time (JIT) compiling and executing Java bytecode [2]. Few classfuzz-generated classfiles can be used for deep JVM testing — most of the mutated classes are rejected by the startup processes. Rarely can the accepted ones be used because the mutated program constructs (variables and their types, methods, *etc.*) are likely dead, *i.e.*, not reached during execution. Additional program constructs, such as methods and call chains, would need to be carefully designed for the mutated constructs to be involved during bytecode verification and execution.

To this end, we introduce *live bytecode (LBC) mutation*, a novel, practical technique for generating valid, executable bytecode from seed classes. The *key insight* behind LBC mutation is to *systematically* manipulate and alter a bytecode file's *live bytecode*, its sequence of executed instructions on a JVM. Deep JVM differences/bugs can then be exposed using the resulting mutants since JVMs may (1) mistakenly analyze a mutant's dataflow during bytecode verification, or (2) verify bytecode with invalid stackmap frames[1] or execute uncommon bytecode instruction sequences in different manners. Two technical difficulties exist in realizing effective LBC mutation.

Difficulty 1: Live Mutant Generation. LBC mutation needs to be designed to generate live mutants for testing JVMs. A simple naïve strategy is to mutate a seed by arbitrarily inserting, deleting, or modifying instructions in its live bytecode. However, this leads to mostly illegal mutants (*i.e.*, those rejected in a JVM's startup process) as Java bytecode needs to satisfy intricate syntactic and structural constraints. For instance, an `invokestatic` instruction should not be inserted unless its first operand refers to a static method, *etc.*

[1]A stackmap frame defines the expected types of local variables and the operand stack of a method during the method's execution [2] [12].

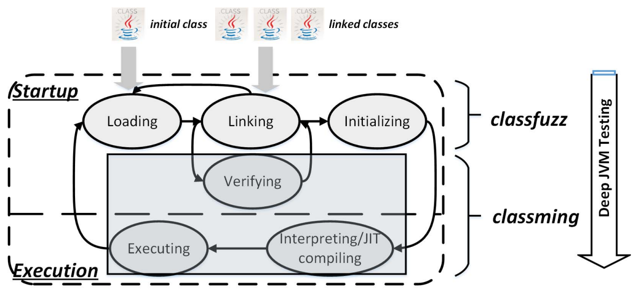

Fig. 1: Simplified process of JVM running classes.

Difficulty 2: Bytecode File Selection. For LBC mutation and effective JVM testing, it is important to select suitable seeds and mutants. In the case of classfuzz (or other coverage-directed fuzz testing techniques, *e.g.*, AFL [13]), it iteratively mutates a seed and retains a mutant if its coverage statistics on the target software system is unique. However, mutants cannot be similarly selected for testing JVMs' execution engines due to non-determinism at runtime. For instance, a JIT compiler allows methods to be compiled in parallel, garbage can be collected as needed, *etc.* All these situations can occur in practice, making the coverage statistics on a JVM's execution engine vary. Thus, coverage uniqueness adopted by classfuzz is unsuited for our purpose.

This paper overcomes these difficulties by introducing classming, a novel realization of LBC mutation for practical differential JVM testing. It operates as follows. First, it records the seed classfile's live bytecode. Second, it systematically manipulates and alters both the control- and data-flow in the seed's live bytecode to generate semantically different mutants from the seed. Finally, the generated mutants are utilized for differentially testing JVMs to expose their differences and thus potential defects.

More importantly, classming takes *an iterative process* of mutant generation. It iteratively generates classfile mutants and selects them using *an acceptance choice*. The choice guides the whole process toward generating live, diverse mutants.

In summary, we make the following main contributions:

- *Objective*. We tackle the challenge of deep validation of production JVMs by testing their bytecode verifiers and execution engines. Different from classfuzz, deep JVM testing requires abundant runnable, diverse test classes. Our work is the first that aims at systematically generating such tests.
- *Approach*. We introduce the LBC mutation approach and a novel, practical realization, classming, for effective differential JVM testing. Given a seed, classming can create a large number of test classes with diverse behaviors to help explore deep JVM differences.
- *Evaluation and defect reporting*. We have evaluated classming on mature JVMs such as OpenJDK's HotSpot and IBM's J9. Using the 14 DaCapo benchmarks as seeds, classming created about 70K test classes, of which more than 1,800 exposed JVM runtime differences and more than 30 triggered JVM crashes. We have analyzed and reported these differences and crashes, of which 14 have already been confirmed as JVM defects and fixed (*e.g.*, dataflow may be incorrectly analyzed, race conditions may prevent JVMs from shutting down cleanly). The IBM Product Security Incident Response Team (PSIRT) has also confirmed a critical security vulnerability in J9 that allowed untrusted code to disable the security manager and elevate its privileges.

II. An Illustrative Example

This section uses a concrete example to motivate and illustrate LBC mutation, which intervenes a class's normal execution by rewriting its executed bytecode. Our realization of LBC mutation, classming, employs the Soot framework [14] [15] [16] to analyze dataflow and bidirectionally transform between a class and its Jimple code (Soot's intermediate representation of a Java class) — transforming a classfile into its Jimple code and dumping a Jimple file into a classfile.

Figure 2a shows the Jimple code of a seed class `seed`. In its `main()` method, `entermonitor r0` (line 8) and `exitmonitor r0` (line 10) denote, respectively, the monitor enter and exit on object `r0`, and synchronize the code block between them. Note that when `entermonitor r0` is executed, `r0` is `null` (line 7).

When the class is run on HotSpot and J9, both JVMs strictly conform to the JVM specification [2] and throw a `NullPointerException`, as the JVM specification states that "*if objectref is null, monitorenter throws a NullPointerException*" (§6.5). The instructions after line 8 are nonlive, that is, the monitor on `r0` is not entered/exited at runtime.

Next, we show three example mutants of `seed`. Each reveals a specific JVM difference when it is run on HotSpot and J9.

Mutant 1. In the first example, classming changes the control flow of the seed class, thus altering its semantics. As Figure 2b shows, two `goto` statements (lines 9 and 15) are inserted into `seed`. It is clear that the resulting class `mutant1` is semantically different from `seed`. In particular, when `entermonitor r0` is executed, `r0` is an initialized object, rather than `null`; `entermonitor r0` will be run for 20 times, while `exitmonitor r0` only once.

When it is run on HotSpot and J9, `mutant1` triggers a difference between the two JVMs — HotSpot throws a runtime exception, while J9 runs the class normally. This difference is caused by the existence of *structured locking*, which occurs during a method invocation when every exit on a given monitor matches a preceding entry on that monitor [2] [17]. If a JVM "*enforces the rules on structured locking*" and if some "*rule is violated during invocation of the current method*," the `return` instruction throws an `IMSE` (`IllegalMonitorStateException`) (§6.5 in the JVM specification). HotSpot enforces the rules on structured locking, throwing an exception because when `main()` returns, the number of monitor entries performed does not equal the number of monitor exits. J9 does not enforce the rules and allows `main()` to return normally.

```
 1  class seed{
 2    public static void main(){
 3      String r0;
 4
 5      r0 = new String;
 6      specialinvoke r0.<String: void init()>();
 7      r0 = null;
 8      entermonitor r0;  //code after is nonlive
 9      ...  //code block to synchronize
10      exitmonitor r0;
11      return;
12
13
14
15  }}
```

(a) Jimple code of a seed class. Both HotSpot and J9 throw a `NullPointerException` for this class.

```
 1  class mutant1{
 2    public static void main(){
 3      String r0;
 4  + int loopcount;
 5  + loopcount = 20;
 6
 7      r0 = new String;
 8      specialinvoke r0.<String: void init()>();
 9  + goto label1;
10      r0 = null;        //skip
11  +label1:
12      entermonitor r0;
13      ...
14  + loopcount = loopcount - 1;
15  + if loopcount>0 goto label1;
16      exitmonitor r0;
17      return;
18  }}
```

(b) Jimple code of `mutant1`. HotSpot throws an `IllegalMonitorStateException`, and J9 runs the class normally.

```
 1  class mutant2{
 2    public static void main(){
 3      String r0;
 4
 5      r0 = new String;
 6  + goto label1;
 7      specialinvoke r0.<String: void init()>();
          //skip
 8      r0 = null;  //skip
 9  +label1:
10      entermonitor r0;
11      ...
12      exitmonitor r0;
13      return;
14  }}
```

(c) Jimple code of `mutant2`. HotSpot runs the class normally, while J9 throws a `VerifyError`.

```
 1  class mutant3{
 2    public static void main(){
 3      String r0;
 4
 5      r0 = new String;
 6      specialinvoke r0.<String: void init()>();
 7  + goto label1;
 8  +label2:
 9      r0 = null;
10  + goto label3:
11  +label1:
12      entermonitor r0;
13      ...
14  + goto label2;
15  +label3:
16      exitmonitor r0;
17      return;
18  }}
```

(d) Jimple code of `mutant3`, which causes HotSpot to throw an `IllegalMonitorStateException` and J9 a `NullPointerException`.

Fig. 2: A seed class and its three classming-generated classes.

Mutant 2. As Figure 2c shows, we create `mutant2` by inserting a `goto` statement into `seed` at line 6. In `mutant2`, when `entermonitor r0` is executed, `r0` is uninitialized.

When `mutant2` is run on HotSpot and J9, it triggers another JVM difference. HotSpot runs the class normally, while J9 throws a `VerifyError` as J9 detects that `r0`, an uninitialized object reference, is monitored.

The HotSpot developers explained that it should be valid to allow an uninitialized reference to be monitored, and there would no potential harm in invoking a monitorenter/exit on an uninitialized instance, although application developers may find the result confusing since the JVM specification states that "*the verifier rejects code that uses the new object before it has been initialized*" (§4.10). At the end, the J9 developers agreed with this explanation, and confirmed the difference to be a defect in J9 and fixed it.

Mutant 3. The final example is `mutant3`. As Figure 2d shows, after three `goto` statements (lines 7, 10, 14) are inserted, the program invokes `monitorenter` on object `r0`, sets `r0` to `null`, and invokes `monitorexit` on `r0`.

HotSpot and J9 throw different exceptions on `mutant3`, which can be propagated if the exceptions are caught and handled respectively. J9 throws a `NullPointerException` at line 16 as the specification states that "*if objectref is null, monitorexit throws a NullPointerException*" (§6.5). HotSpot, on the other hand, makes a rarely known optimization (even to HotSpot developers): when the method completes abruptly and the rules on structured locking are violated, the JVM throws an `IMSE`, and disposes the other exceptions.

Summary. The three illustrative examples clearly demonstrate the strength of LBC mutation. Mutating a seed class, in particular, changing its control- and data-flow, can help create mutants with diverse semantics. When run on multiple JVM implementations, these mutants may expose JVM behavior differences and thus potential defects. Note that some differences can only be exposed by classes generated via two or more iterations of mutation, assuming that each iteration inserts one `goto` statement. Our realization classming provides an iterative process in which live mutants can be continuously created for differential JVM testing.

III. Approach

classming takes an iterative process to generate test bytecode files. As Figure 3 shows, given a seed bytecode file f, classming aims at creating a mutant of f by manipulating f's live bytecode (Section III-A), and iteratively creates and selectively accepts f's mutants (Section III-B) — during each iteration, *an acceptance choice* chooses either the resulting mutant or its seed for the next iteration.

The generated classfile mutants are then employed to differentially test JVMs and uncover differences. Any JVM differences, if exposed, become oracles for finding flaws in the tested JVMs. Interesting differences include (1) *JVM crashes*, (2) *verification differences* (*e.g.*, JVMs verify `mutant2` in Section II differently), and (3) *execution differences* (*e.g.*, HotSpot and J9 throw different exceptions for `mutant1`).

A. Live Bytecode Mutation

Live bytecode of a classfile f refers to, when f is run on a JVM, a sequence of executed bytecode instructions $\mathscr{L} : [I_0, I_1, \dots, I_m]$. Similarly, we use *live mutants* and *live methods* to denote those that can be executed at runtime.

An *LBC mutation* is performed on f by instrumenting a *hooking instruction* (*HI*) at a program point before $I_{0 \le i \le m}$. As Figure 4 shows, a hooking instruction hijacks control flow, and alters f's execution to $\mathscr{L}' : [I_0, I_1, \dots, I_{i-1}, I_j, \dots, I_n]$. Here we call the program point after I_{i-1} a *hooking point* (*hp*), and that before I_j a *target point* (*tp*). The resulting mutant, say g (with the live bytecode instructions $\mathscr{L}'$), can become a corner case for triggering JVM differences, since it may contain abnormal stackmaps or unusual dataflow that can confuse a JVM or even crash it at runtime.

Algorithm 1 shows a process of LBC mutation, including three main steps: (1) select LBC mutators, (2) select methods to mutate, and (3) insert hooking instructions.

Step 1: Select LBC Mutators. We leverage the five Soot's Jimple instructions ((1) **`goto`**, (2) **`return`**, (3) **`throw`**, (4) **`lookupswitch`**, and (5) **`tableswitch`**) as HIs — Soot uses 15 Jimple instructions and only five can alter a program's control-flow. As LBCMutation() in Algorithm 1 shows, an LBC mutation picks up an HI by random and instruments it into the seed classfile, altering the control- and data-flow at runtime (*e.g.*, enforcing the execution to jump, an exception to be thrown, or an invoked method to return).

```
loopcount = M; // M is a positive integer
...
label0: //insert a label at a target point
...
loopcount = loopcount - 1;
if loopcount>0 goto label0; //insert an HI at a
    hooking point
...
```

Each HI, as shown in the above code segment, is supplemented with a condition (*e.g.*, `if loopcount>0`), which is to bound the possible iterations for each introduced loop. This avoids the occurrences of infinite loops as well as enables additional altering of control-flow.

An instrumented HI can also be deleted. It helps create mutants with diverse HI combinations or prevent them from getting stuck (*e.g.*, a mutant may be run slowly if it contains a computation-intensive loop).

Step 2: Select methods to mutate. During each iteration, classming selects one class method to mutate. Random selection offers no guidance — some methods need to be more frequently mutated than others because of their more complex structures and richer instructions. Thus, we select methods by a potential function:

$$potential(m) = \frac{\#inst}{\#mutation},$$

where a method m's potential relies on its size (*#inst*) and how many times it has already been mutated (*#mutation*).

Intuitively, the higher a method's potential, the more likely it needs to be mutated. A method's potential decreases after it is mutated. To capture this intuition, we choose methods with probabilities meeting a geometric distribution that allows methods with higher potentials to be selected with higher probabilities.

Let M denote an array where all the live methods in f are stored and sorted in descending order of their potentials. Let *size* denote the size of M. Let the k^{th} method in M be chosen with probability $(1-p)^k p$, where p is the success probability. Obviously, the sum of the probabilities needs to reach 1, *i.e.*, $1-(1-p)^{(size-1)+1} \approx 1$. Here we let $(1-p)^{size} = \varepsilon$, where ε is a very small value (*e.g.*, 0.05).

A method can then be chosen from M using *rand*, a random real value between 0.0 and 1.0. The k^{th} method is selected to be mutated when

$$1-(1-p)^k \le r < 1-(1-p)^{k+1}$$

Since k is an integer denoting an array index, we have

$$k = \lfloor log_{\varepsilon}(1-rand)^{size} \rfloor,$$

indicating that method selection relies only on *rand*.

Step 3: Insert hooking instructions. LBC mutation alters a program's dataflow, which is carefully analyzed by JVMs during bytecode verification. Here, dataflow is mainly introduced via data dependencies established among variable definitions (*defs*) and uses (*uses*) [18] [19].

• ***Select hooking points***. An HI deliberately destroys a seed's data dependencies for creating corner cases. To simplify our discussion, we assume that, after an HI is instrumented, all data dependencies (*i.e.*, `def-use`, `def-def`, `use-use`, and `use-def`) that pass through the hooking point will be intercepted, which may result in abnormal data-flow (*e.g.*, an object is *redefined* or *undefined*). The more data dependencies are intercepted, the more likely the mutant contains abnormal dataflow that a JVM may fail to analyze.

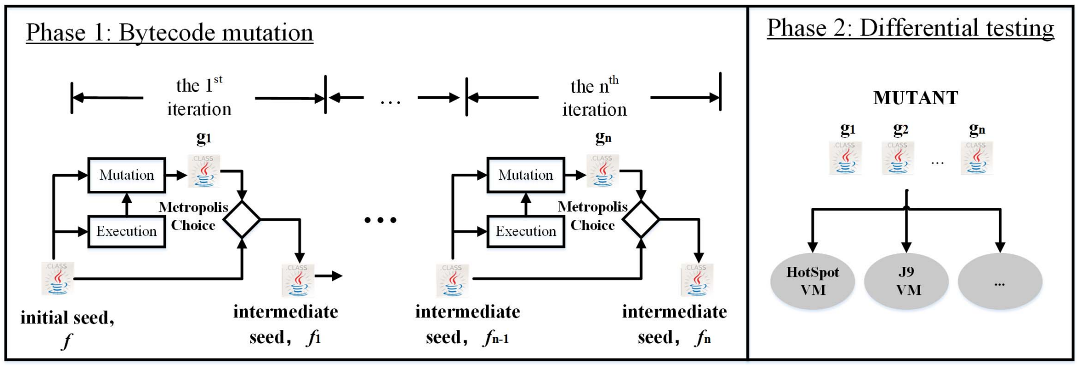

Fig. 3: An overview of the classming approach.

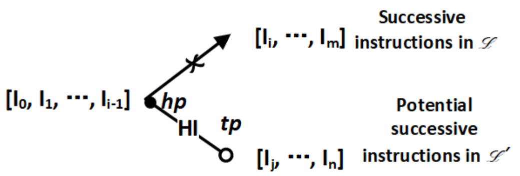
[I0, I1, ···, Ii-1]
hp
HI
tp
Successive
[Ii, ···, Im] instructions in ℒ
Potential successive
[Ij, ···, In] instructions in ℒ'

Fig. 4: Mutating a seed file with live bytecode $\mathscr{L}$ to a mutant with $\mathscr{L}'$.

In order to intercept as many data dependencies as possible, classming selects a hooking point judiciously during each iteration. As SELECTHOOKINGPOINT() in Algorithm 1 shows, classming randomly chooses n program points as candidates (in this algorithm, $n = 2$). It then calculates the potential data dependency interceptions *w.r.t.* these candidates respectively (lines 23 $\sim$ 32), and chooses the one with the most interceptions as a hooking point (line 33). Existing data dependencies are obtained by analyzing the seed's live bytecode.

• ***Select target points***. An LBC mutation can require one or more labels to be inserted into the bytecode, along with an insertion of a `goto` or a `switch` instruction.

In order to generate a mutant semantically different from the previously generated mutants, classming favors inserting labels before instructions missed from previous runs. As SELECTTARGETPOINT() in Algorithm 1 shows, it chooses at random a target point candidate, say *tp*, and (1) accepts *tp* if the successive instruction (say *I*) has never been hit in the previous runs, (2) accepts it with a high probability (*e.g.*, 0.8) if *I* is not hit by the last mutant, or (3) accepts it with a low probability (*e.g.*, 0.2). This strategy allows seed coverage, a fitness function for guiding mutant acceptance (see Section III-B), to increase during the mutation process.

B. Mutant Acceptance

As explained in Section I, distinct coverage statistics *w.r.t.* a JVM cannot be used to accept classfile mutants. Thus classming employs *seed coverage* as the fitness function, and adopts *the Metropolis-Hastings (MH) algorithm* [20] [21] for accepting mutants during the mutation process.

Seed coverage. Seed coverage is a metric denoting how many instructions in the initial seed are covered by its mutants, as the fitness function for accepting mutants for further mutations. This fitness function directs the mutation process in two respects: (1) Mutants with different seed coverage values have different semantics (runtime behaviors), and (2) Live mutants can be more easily created from (intermediate) seeds with higher seed coverage.

To this end, we record the live bytecode of each mutant. Assume f has x instructions, y of which can be hit when a mutant, say g, is run on a JVM. The seed coverage of g *w.r.t.* f is calculated by

$$cov_{seed}(g) = \frac{y}{x} \times 100\%$$

A sampling process. classming adopts the MH algorithm, an MCMC (Markov Chain Monte Carlo) sampling method, for selecting mutants as intermediate seeds. The MH algorithm aims to accept a sequence of samples whose distribution closely approximates the desired distribution [20] [21].

In our setting, we let each sample be the seed coverage of a mutant, and let the exponential distribution be the desired distribution. The desired distribution facilitates choosing of mutants with high seed coverage for further mutations, since these mutants have the more vitalities than those with low seed coverage. A mutant with low seed coverage may easily lead to cases that cannot be further mutated.

For MCMC sampling, we begin with a transformation of the fitness function into a probability density function [22] [23]

$$P(f) = \frac{1}{Z} exp(-\beta(cov_{seed}(f)))$$

where β is a constant (in our setting $\beta = 0.08 \times x$), and Z a partition function that normalizes the distribution.

We assume that the proposal distribution is symmetric. classming uses Metropolis choice for sampling acceptance [22]

$$\begin{aligned} A(f \to g) &= min(1, \frac{P(g)}{P(f)}) \\ &= min(1, exp(\beta(cov_{seed}(f) - cov_{seed}(g))) \end{aligned}$$

The Metropolis choice directs the mutation process. As Algorithm 2 shows, we run and collect the seed coverage of each mutant, and accept live, diverse mutants using the acceptance probability (lines 9 $\sim$ 14); we reject the nonlive

Algorithm 1 LBC Mutation

```
Input: $f$: a seed classfile; $livecodeset$:$\{\mathscr{L}_0, \ldots, \mathscr{L}_n\}$, a set of live bytecode
    of the previously generated mutants
Output: $g$: a classfile mutant
 1: function LBCMUTATION($f, livecodeset$)
 2:     $method \leftarrow$ SELECTMETHODTOMUTATE($\mathscr{L}_n$)
 3:     $t \leftarrow$ select an LBC mutator by random
 4:     if $t$ is to instrument a goto instruction then
 5:         $hp \leftarrow$ SELECTHOOKINGPOINT($method, \mathscr{L}_n$)
 6:         insert if (...) goto newlabel; at $hp$
 7:         $tp \leftarrow$ SELECTTARGETPOINT($method, livecodeset$)
 8:         insert newlabel: at $tp$
 9:     else if $t$ is to instrument a lookupswitch or a tableswitch
    instruction then
10:         ...// similar to goto but with multiple target points
11:     else if $t$ is to instrument a return or a throw instruction then
12:         ...// similar to goto but without any target points
13:     else
14:         ...// select from $method$ a live HI and remove it
15:     $g \leftarrow$ UPDATE($f$) // replace $method$ in $f$ with its mutant
16:     return $g$

17: function SELECTMETHODTOMUTATE($\mathscr{L}_n$)
18:     $M \leftarrow$ GETLIVEMETHODS($\mathscr{L}_n$)
19:     $M \leftarrow$ SORT($M$) // sort methods using their potentials
20:     $rand \leftarrow$ NEW RAND()
21:     return $M$.GET($\lfloor log_{\varepsilon}(1 - rand)^{size} \rfloor$)

22: function SELECTHOOKINGPOINT($method, \mathscr{L}_n$)
23:     let the live bytecode on $method$ be $[I_0, I_1, \ldots, I_m]$
24:     // choose $\alpha$ and compute $\gamma_\alpha$ (i.e., the data dependency interceptions
    w.r.t. $\alpha$)
25:     let $\alpha$, a program point after $I_{i-1}$, be chosen
26:     set $s_0 \leftarrow s_1 \leftarrow \emptyset$
27:     for $I : [I_0, \ldots, I_{i-1}]$ do              // DEF($I$) and USE($I$) compute the
28:         $s_0 \leftarrow s_0 \cup$ DEF($I$) $\cup$ USE($I$)     // variables defined and used in $I$,
29:     for $I : [I_i, \ldots, I_m]$ do                  // respectively
30:         $s_1 \leftarrow s_1 \cup$ DEF($I$) $\cup$ USE($I$)
31:     set $\gamma_\alpha \leftarrow s_0 \cap s_1$
32:     ...// choose $\beta$ and compute $\gamma_\beta$
33:     return $(\gamma_\alpha.size \geq \gamma_\beta.size ? \alpha : \beta)$

34: function SELECTTARGETPOINT($method, livecodeset$)
35:     while true do
36:         select $tp$, a target point candidate before an instruction $I$ in $method$
37:         if $I \notin (\mathscr{L}_0$.ASSET() $\cup \cdots \cup \mathscr{L}_n$.ASSET()) then return $tp$
38:         $rand \leftarrow$ NEW RAND()
39:         if $I \notin \mathscr{L}_n$.ASSET()&& $rand < prob_{high}$ then return $tp$
40:         if $I \in \mathscr{L}_n$.ASSET()&& $rand < prob_{low}$ then return $tp$
```

Algorithm 2 Iteratively creating and selectively accepting mutants

```
Input: $f$: a seed bytecode file; $iter$: the number of iterations
Output: set $MUTANT$, $ACC$, $REJ$, $NONLIVE$
    /*$MUTANT = ACC \cup REJ \cup NONLIVE$ contains classfile mutants for
    JVM testing, $ACC$ and $REJ$ contain live ones that are accepted and
    rejected, respectively, and $NONLIVE$ contains nonlive ones rejected by
    a JVM's startup process.*/
 1: $MUTANT \leftarrow ACC \leftarrow REJ \leftarrow NONLIVE \leftarrow \emptyset$
 2: set $livecodeset \leftarrow \{f.livecode\}$
 3: for $i : 1$ to $iter$ do
 4:     $g \leftarrow$ LBCMUTATION($f, livecodeset$)
 5:     if ($g$ is created successfully) then
 6:         $MUTANT \leftarrow MUTANT \cup \{g\}$
 7:         if $cov_{seed}(g) > 0$ then
 8:             $rand \leftarrow$ NEW RAND()
 9:             if $rand < A(f \rightarrow g)$ then
10:                 $ACC \leftarrow ACC \cup \{g\}$
11:                 $livecodeset \leftarrow livecodeset \cup \{g.livecode\}$
12:                 $f \leftarrow g$
13:             else
14:                 $REJ \leftarrow REJ \cup \{g\}$
15:         else
16:             $NONLIVE \leftarrow NONLIVE \cup \{g\}$
17: return $MUTANT$, $ACC$, $REJ$, $NONLIVE$
```

bytecode files (line 16), since it is unlikely to obtain a live bytecode file through mutating a nonlive seed.

All the generated mutants (including those accepted and rejected, and the nonlive ones) will be employed as tests in differential JVM testing.

IV. Evaluation

We evaluated classming on two mainstream JVM implementations, *i.e.*, HotSpot and J9. Our evaluation is aimed at answering three research questions:

- ***RQ1:*** Can classming generate sufficient valid test bytecode files for JVM testing?
- ***RQ2:*** How effective are the classming-generated mutants in testing JVMs?
- ***RQ3:*** What are the root causes of detected JVM differences?

A. Evaluation Setup

Approaches for comparison. We compared classming with the following approaches to investigate the benefits of each of classming's strategies.

- classfuzz' is a mutant of classfuzz [10]. It alters classfiles using six instruction-level mutators provided by classfuzz (*i.e.*, *inserting*, *replacing*, *deleting*, *exchanging*, *duplicating*, and *cloning* bytecode instructions). Note that classfuzz itself was not included in our evaluation since it does not aim for deep JVM testing.
- clrandom manipulates at random control- and data-flow.
- clgreedy employs a greedy strategy in accepting mutants — a mutant is accepted only when it allows the accumulative seed coverage to increase.

The table below summarizes the differences of the four evaluated approaches, including the mutators and the strategies for creating and accepting mutants.

Approach	Mutators	How to create a mutant?	How to accept a mutant?
classming	HI insertions/deletions	LBC mutation	The MH algorithm
classfuzz'	6 classfuzz mutators	LBC mutation	The MH algorithm
clrandom	HI insertions/deletions	Random mutation	The MH algorithm
clgreedy	HI insertions/deletions	LBC mutation	A greedy algorithm

Benchmarks and Configurations. We mutated the DaCapo benchmarks, a collection of open-source, real-world applications, such as `eclipse`, `lusearch`, and `pmd` [24].

As Table I shows, for each benchmark, classming mutated its initial classfile, creating mutants of this benchmark. Note the non-initial classfiles were not used as seeds, as they are not definitely linked at runtime. For each seed, the number of iterations (#*iter*) was set about 14× of the lines of Jimple instructions (#*inst*), allowing the seed to be sufficiently mutated. These benchmarks have been mutated for 88K times.

HotSpot (Java 9) was used as the reference JVM for generating classfile mutants and collecting coverage statistics. HotSpot (build 9-ea+172) and J9 (build 2.9) were chosen for

TABLE I: Seeds, iteration times, and numbers of test classfiles generated. In this table, we use V, L and N to denote the numbers of generated mutants, live mutants and nonlive ones, respectively. Here we have $V = L + N$.

Benchmark	Initial class (seed)	*#inst*	*#iter*	classming				classfuzz'				clrandom				clgreedy			
				V	L	N	$sbr(\%)$	V	L	N	$sbr(\%)$	V	L	N	$sbr(\%)$	V	L	N	$sbr(\%)$
`avrora`	`.../avrora/Main.class`	279	5000	4105	3063	1042	17.9	2534	2243	291	49.3	4512	3638	874	9.8	3997	2787	1210	20.1
`batik`	`.../rasterizer/Main.class`	428	8000	6511	4421	2090	18.6	3140	2823	317	60.7	7115	5368	1747	11.1	6451	4442	2009	19.4
`eclipse`	`.../EclipseStarter.class`	2005	20000	16405	8448	7957	18.0	8749	7805	944	56.3	17581	11081	6500	12.1	18113	13167	4946	9.4
`fop`	`.../fop/cli/Main.class`	174	3000	2484	2271	213	17.2	1203	977	226	59.9	2735	2678	57	8.8	2317	1599	718	22.8
`h2`	`.../h2/TPCC.class`	1022	15000	12102	7559	4543	19.3	5861	5114	747	60.9	13244	9729	3515	11.7	13034	8748	4286	13.1
`jython`	`.../python/util/jython.class`	351	6000	4851	2784	2067	19.1	2615	2316	299	56.4	5531	4348	1183	7.8	5159	2536	2623	14.0
`luindex`	`.../luindex/Index.class`	74	2000	1688	1198	490	15.6	1100	968	132	45.0	1794	1229	565	10.3	1677	1112	565	16.1
`lusearch`	`.../lusearch/Search.class`	144	2500	2102	1495	607	15.9	1453	1228	225	41.9	2380	1965	415	4.8	2076	1123	953	17.0
`pmd`	`.../pmd/PMD.class`	821	10000	8261	4725	3536	17.4	3191	2931	260	68.1	9042	6307	2735	9.6	8337	4334	4003	16.6
`sunflow`	`.../sunflow/Benchmark.class`	248	4000	3142	2219	923	21.4	1141	1020	121	71.5	3502	2987	515	12.4	3238	1903	1335	19.0
`tomcat`	`.../tomcat/Control.class`	42	2000	1732	1515	217	13.4	1078	905	173	46.1	1885	1672	213	5.7	1745	1348	397	12.7
`tradebeans`	`.../daytrader/Launcher.class`	251	4000	3161	1833	1328	21.0	1884	1530	354	52.9	3441	2210	1231	14.0	3487	2511	976	12.8
`tradesoap`	`.../daytrader/Launcher.class`	251	4000	3200	1972	1228	20.0	1979	1676	303	50.5	3436	2166	1270	14.1	3346	1830	1516	16.3
`xalan`	`.../xalan/XSLTBench.class`	129	2500	2101	1364	737	16.0	1235	1093	142	50.6	2291	1689	602	8.4	2233	1297	936	10.7
Total		**6219**	**88000**	**71845**	**44867**	**26978**	**18.4**	**37163**	**32629**	**4534**	**57.8**	**78489**	**57067**	**21422**	**10.8**	**75210**	**48737**	**26473**	**14.5**

differential testing. The evaluation was run on the Ubuntu 16.04 machines with Intel Core i7-6700 CPU@3.40HZ and 8GB RAM. The time of executing each mutant was constrained within 20 seconds.

Metrics. We repeated each approach five times and chose the test suite with the most mutants for comparison. Four metrics were taken to evaluate classming against the other approaches.

• ***Stillborn rate*** To quantitatively measure whether an approach can generate sufficient valid tests, we computed the *stillborn rate* of each approach

$$sbr = 1 - \frac{|MUTANT|}{\#iter} \times 100\%$$

where $MUTANT$ (*cf.* Algorithm 2) is the test suite for JVM testing, and $|MUTANT|$ is its size.

This rate is widely-used in mutation testing to evaluate the applicability and effectiveness of mutant generation techniques [25] [26]. In this paper, *stillborn mutants* are Jimple files that are syntactically invalid and thus cannot be transformed into test classfiles. The higher the rate is, the more stillborn mutants are created.

• ***Accumulative seed coverage*** To measure whether a seed has been fully exploited for generating mutants, we computed the *accumulative seed coverage w.r.t.* a test suite $MUTANT$: $\{g_1, g_2, \ldots\}$ by running the mutants on a JVM (*e.g.*, HotSpot)

$$asc = cov_{seed}(g_1) \oplus cov_{seed}(g_2) \oplus \ldots$$

where $\oplus$ is an operator that merges two mutants' (*e.g.*, g_1 and g_2) seed coverages. Intuitively, the higher the coverage, the more thoroughly the original seed is exploited by its mutants.

• ***JVM code coverage*** The seed coverage and JVM's code coverage are combined to direct exploring the input space and triggering JVMs' functionalities (*e.g.*, error handling). Thus it is still interesting to investigate how the coverage is improved by the mutants.

We ran the seeds and the classming test suites on HotSpot and collected their JVM statement coverage statistics (say $Jcov$). We computed the coverage increment achieved by each test suite $MUTANT$, using

$$Jinc = Jcov(MUTANT) \oplus Jcov(seed) - Jcov(seed),$$

where $\oplus$ merges the coverage statistics.

• ***JVM differences*** We counted the JVM differences found by each approach, analyzed each difference we found, and reported any potential defects to JVM developers.

B. *RQ1: Sufficiency of classming-generated mutants*

Stillborn rate. Table I compares the sizes of the test suites. In $88K$ mutations, classming, clrandom, and clgreedy generated $71,845$, $78,489$, and $75,210$ bytecode files, with the stillborn rates of 18.4%, 10.8%, and 14.5%, respectively. In contrast, classfuzz' generated $37,163$ bytecode files, with a stillborn rate of 57.8% — about 40% higher than those of the other approaches.

Table I clearly demonstrates that the LBC mutators taken by classming, clrandom, and clgreedy can help generate many more test bytecode files than the mutators taken by classfuzz'. The main reason is that bytecode mutation is performed on Jimple code. classfuzz' manipulates a Jimple file by inserting and/or deleting some arbitrary instructions, which frequently results in invalid Jimple files that cannot be transformed into classfiles. The LBC mutators, on the other hand, are less likely to destroy the file's syntactical integrity, although $10.8 \sim 18.4\%$ of the resulting Jimple files still violate Soot's constraints (*e.g.*, a method must return a value if it has a return type, a label cannot be inserted before variable declarations, *etc.*).

These results lead to our first finding:

Finding 1: classming achieved a stillborn rate nearly 40% lower than that of classfuzz'; the LBC mutators allow test bytecode files to be sufficiently generated.

C. *RQ2: Effectiveness of classming-generated mutants*

Accumulative seed coverage. Table II shows the seed coverage achieved by the test suites. Let the DaCapo benchmarks be used as the baseline for measuring coverage improvement. When the seeds were run, $2,852$ of $6,219$ statements were covered. When the mutants were run, the seed coverage could be improved by $15 \sim 31\%$. The improvement on seed coverage is obvious. These approaches take mutators that can alter control-flow and employ seed coverage as a guidance, allowing nonlive bytecode instructions to be run and the accumulative seed coverage to increase. However, many test suites could not achieve 100% of seed coverage, as some seed methods may not be reachable.

Furthermore, the classming/clrandom test suites obtained higher seed coverage than the clgreedy ones, demonstrating

TABLE II: Seed coverage achieved by the test suites and JVM differences uncovered by the four approaches. Here each number inside "()" denotes the number of unique execution differences.

Benchmark	*#inst*	*asc*					**classming**			**classfuzz'**			**clrandom**			**clgreedy**		
		baseline	classming	classfuzz'	clrandom	clgreedy	#crashes	#exec.diffs	#verif.diffs	#crashes	#exec.diffs	#verif.diffs	#crashes	#exec.diffs	#verif.diffs	#crashes	#exec.diffs	#verif.diffs
avrora	279	0.51	0.95	0.86	0.94	0.79	2	6(6)	0	3	0	0	0	0	0	0	0	0
batik	428	0.81	0.97	0.96	0.97	0.82	29	-	0	0	-	0	0	-	0	0	-	0
eclipse	2005	0.43	0.75	0.58	0.72	0.49	0	-	8	0	-	0	0	-	0	0	-	0
fop	174	0.16	0.24	0.23	0.21	0.24	0	0	24	0	0	0	0	0	0	0	0	0
h2	1022	0.32	0.73	0.5	0.68	0.59	0	-	0	0	-	0	0	-	0	0	-	0
jython	351	0.3	0.85	0.34	0.5	0.45	0	4(2)	0	0	4(2)	0	0	1453(3)	0	0	0	0
luindex	74	0.82	1	0.99	1	0.99	0	0	0	0	0	0	0	0	0	0	0	0
lusearch	144	0.81	0.98	0.88	0.97	0.88	0	-	0	0	-	0	0	-	0	0	-	0
pmd	821	0.38	0.67	0.63	0.66	0.56	0	-	0	0	-	0	0	-	0	0	-	0
sunflow	248	0.2	0.53	0.47	0.55	0.69	0	1805(31)	0	0	56(5)	0	0	2227(19)	0	0	107(1)	0
tomcat	42	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
tradebeans	251	0.77	0.94	0.93	0.89	0.85	0	-	0	0	-	0	0	-	0	0	-	0
tradesoap	251	0.65	0.92	0.84	0.92	0.86	0	-	0	0	-	0	0	-	0	0	-	0
xalan	129	0.76	1	0.79	1	1	0	-	0	0	-	0	0	-	0	0	-	0
Total	**6219**	**0.46**	**0.77**	**0.63**	**0.73**	**0.61**	1878(102)			63(10)			3680(22)			107(1)		

that the MH algorithm works better than the greedy algorithm in exploring these seeds. The greedy algorithm usually fails in constructing live bytecode for hitting deep instructions. These results lead to our second finding:

Finding 2: The classming test suites achieved higher seed coverage than the clgreedy ones; the MH algorithm enables more effective exploration of the mutant space *w.r.t.* a seed than the greedy algorithm.

JVM code coverage. In real circumstance, it is expensive to run all of the mutants and collect their JVM code coverage. Thus we picked up the last 100 classfiles in each classming test suite to compose a mini test suite and ran it to collect JVM code coverage statistics.

In the evaluation, the initial classfiles (*i.e.*, the seeds) covered $117,744 \sim 135,118$ lines of JVM's source code. Comparatively, the mini test suites covered $112,660 \sim 136,095$ lines of code, with increments of $694 \sim 8555$ lines (3,668 lines on average). It denotes that the mutants supplement the seeds in covering additional JVM's source code during testing. Although the coverage increments appear relatively low (considering that HotSpot has 260K lines of code), the classming approach is promising in exploring JVM's code because only the initial classfiles were mutated and only a small number of mutants were run for coverage analysis. Additional seeds and techniques may be used to maximize JVM's code coverage, which makes interesting future work.

Table III enumerates the top 10 source packages that contribute to the coverage increment (*i.e.*, 3,668 lines on average). The source packages *w.r.t.* the optimizing JIT compiler ("`opto`"), the shared objects ("`libjvm/objs`"), and the bytecode verifier ("`classfile`") contribute to 72.1% ($= \frac{2603}{3608} \times 100\%$) of the coverage increment. The results lead to our third finding:

Finding 3: The classming mutants indeed facilitate deep testing of JVMs' bytecode verifiers and execution engines.

JVM differences. We used the test suites to differentially test JVMs and summarized the JVM differences in Table II. In this table, the accepted and rejected test classfiles generated from the benchmarks "`batik`", "`eclipse`", "`h2`", "`lusearch`", "`pmd`", "`tradebeans`", "`tradesoap`", and "`xalan`" were not used in differential JVM testing. The main reason for this is that user threads exist in these seed classfiles and differences, if revealed, may not expose JVM defects.

The results clearly demonstrate that classming and clrandom uncover more JVM differences/crashes than classfuzz' and clgreedy. In addition, only classming-generated test suites revealed verification differences, indicating classming is the most effective in creating classfiles having abnormal dataflow.

However, the numbers of execution differences in Table II can be bloated: If a seed can trigger an execution difference, its mutants tend to trigger the same type of differences. Thereafter, we counted the *unique execution differences* in this way: Let g be a mutant mutated directly from f. Let g trigger an execution difference (say $diff$) when it is run on HotSpot and J9. $diff$ is *unique* if f cannot trigger $diff$. Specifically, we find that (1) most of the execution differences uncovered by clrandom were redundant, and (2) the classming test suites triggered $10\times$ and $4.6\times$ as many unique differences as those uncovered by the classfuzz' and clrandom test suites, respectively. The results lead to our fourth finding:

Finding 4: Only the classming test suites revealed verification differences; they also exposed $10\times$ and $4.6\times$ as many unique JVM differences as those exposed by the classfuzz' and clrandom test suites, respectively;

D. *RQ3*: *Difference analysis and bug report*

We analyzed and reported a number of JVM differences to the JVM developers, 14 of which have been confirmed as JVM defects and/or fixed. We summarize below some typical defects and differences.

Security vulnerability in J9. The IBM Product Security Incident Response Team (PSIRT) has confirmed a critical security vulnerability in J9 that was easy to exploit, and allowed untrusted code to disable the security manager and elevate its privileges (a CVE with a CVSS base score 9.8). Next,

TABLE III: Top 10 HotSpot's source packages that contribute to the coverage increment.

package	**main use**	*Jinc*
`opto`	Opto compiler (*i.e.*, the C2 compiler, a highly optimizing bytecode compiler)	1271
`libjvm/objs`	Shared objects (in libjvm.so)	757
`classfile`	Manipulation of Java classes	575
`runtime`	VM's runtime management	211
`cpu/x86/vm`	Facilities for supporting the execution of JVM on x86 cpus	201
`c1`	C1 compiler	114
`ci`	Internal JVM compiler interfaces	89
`oops`	Manipulation of objects	86
`utilities`	HotSpot's source utilities	70
`code`	Management of code stubs	54

we present a simplified mutant that can expose this security vulnerability.

```
 1  abstract interface A extends java.lang.Object{
 2      public void go() ;
 3  }
 4  class Search extends java.lang.Object implements
        A {
 5      public void go()     {
 6          Search $r2;
 7          $r2 := @this: Search;
 8          ...//J9 allows to use $r2 here
 9          return;
10      }
11      public static void main(){
12          Search $r2;
13          $r2 = new Search;
14  + goto label1:
15          ...//$r2 is initialized here
16  +label1:
17          interfaceinvoke $r2.<A: void go()>();
18  } }
```

In this program, an insertion of an HI (line 14) leads to a use of an uninitialized object, `$r2`, at line 17. J9 can run the class `Search` normally, which clearly indicates that its verifier fails to reject code that uses an object before it has been initialized.

More seriously, this program shows that `$r2` can be transferred, through invoking an interface method `go()` (line 17), outside of `main()`. Since a verifier verifies bytecode only at the method level, assuming that all of the arguments transferred into a method have been initialized, the verifier also fails to reject `go()`, even if `$r2` is used in it. The JVM instruction `invokeinterface` (corresponding to the Jimple instruction `interfaceinvoke`) becomes a backdoor that allows uninitialized objects to be transferred and used in other methods, incurring high risks.

Defects in bytecode verifiers. J9's verifier may miscalculate dataflow, making it incorrectly accept or reject bytecode files. For example, J9 incorrectly throws verifier errors due to miscalculating local variables in stackmap frames. J9 developers fixed the defect using a mutant of `fop`. The patch is to ensure that the verifier checks long/double type only when there are still local variables left in the stackmap frames.

J9 may incorrectly verify code with uninitialized objects. For example, Section II shows that J9 can mistakenly reject `monitorenter/monitorexit r0` when `r0` is uninitialized.

In some extreme cases, J9 can make mistakes when verifying code segments containing instructions such as `if_acmpeq`, `if_acmpne`, `ifnull`, `ifnonnull`, and `aastore`. For example, J9 verifies the two code segments below differently, but they are in fact semantically equivalent (corresponding to `if (this==o)` and `if (o==this)`).

```
/*J9 may verify the next two bytecode segments
    differently, while HotSpot verifies them
    consistently.*/
(1) 0: new         #2    (2) 0: aload_0
    3: aload_0                1: new        #2
    4: if_acmpeq  7           4: if_acmpeq  7
```

JVM crashes. J9 developers have fixed a race condition using a small test suite reported by us. J9 can crash frequently, with a report indicating there exists a double free problem. J9 developers confirmed the cause of the crash to be a flaw in the JIT compiler that prevented the JVMs from shutting down cleanly. The IBM developers also communicated to us that they used this test suite to expose a VM issue happening on old Linux kernels.

J9 may crash when generating the messages for operand stack underflows. Its developers found that it would be too late to check stack underflow as they did before. They fixed J9 by enforcing extra checking of stack underflow. In addition, J9 creates crash-dump files when throwing `OutOfMemoryErrors`, while HotSpot does not.

Other execution differences. The execution differences revealed by the `jython` mutants (see Table II) have been eliminated since OpenJDK (build 9+181) was released. In addition, those revealed by the `sunflow` mutants were raised due to certain non-determinism in the mutants.

V. Related Work

We discuss two strands of related work: JVM testing and mutation-based fuzz testing.

JVM testing. A number of test suites and benchmarks have been designed for JVM testing. Among them, the suite most widely-used is the Java Compatibility Kit (JCK) [28], an extensive test suite provided by Oracle to ensure the compatible implementation of the Java platform. JVMs are required to meet JCK in case of any changes or fixes. Many other Java benchmarks, such as the DaCapo benchmark suite [24], SPECjvm2008 [29], SPECjbb2013 [30], SciMark 2.0 [31], CD_x [32], and Stanford SecuriBench [33], have been developed for different purposes. Despite their importance in testing and regression testing, these tests were not designed to expose defects in released JVMs. Instead, classming allows a large number of live tests to be created from existing classes and applications, significantly enhancing JVM testing.

Efforts have been spent on automated test generation for JVM testing. Sirer and Bershad propose lava, enabling users to use a production grammar to produce test classes [34]. Yoshikawa *et al.* propose to generate classes by producing a class's control flow and filling the bytecode into control flow edges [35]. Freund and Mitchell introduce a type system that can be applied to generate faulty classes and look for inconsistencies among bytecode verifiers [12]. Calvagna *et al.* model a JVM as a finite state machine for deriving classes [36]–[38]. Compared with these techniques, classming generates test

TABLE IV: Technical comparison among classming, classfuzz [10], and Hermes [27].

Technical difference	classming	classfuzz	Hermes
1. The general process of test generation			
1.1 *Process*	An iterative mutation process	An iterative mutation process	A random mutation process
1.2 *How is the mutation process directed?*	Mutants are selectively accepted.	Mutators are selectively employed.	/
2. Each iteration in the mutation process			
2.1 *Mutation objective*	The resulting mutant is live and semantically different from its seed.	The JVM's startup processes for the resulting mutant and its seed are different.	The resulting mutant is semantically equivalent to its seed *w.r.t.* provided inputs.
2.2 *Type of seed/mutant*	Java bytecode file	Java bytecode file	C source code
2.3 *Mutators*	Inserting/deleting HIs for manipulating control- and data-flow in the seed	129 mutators for mutating Java bytecode syntactically	Inserting EMI snippets (FCB, TG, and TCB)
2.4 *What needs to be altered?*	Live bytecode	Any construct of the seed	Live and dead code regions
2.5 *How is the resulting mutant accepted?*	Mutant has high seed coverage and is semantically different from the seed.	Mutant is different from the seed in their JVM's code coverage.	Mutant is an EMI mutant of its seed.
3. Testing/differential testing			
3.1 *Test subjects*	JVMs' verifiers and execution engines	JVMs' startup processes	C Compilers
3.2 *Test oracle*	Verification/execution differences among JVMs	Differences among JVMs' startup processes	Inconsistent outputs between a seed and its EMI mutant

classfiles by manipulating live bytecode of existing classfiles, rather than leveraging grammars or formal models.

Similar to classming, classfuzz [10], Java* Fuzzer [39], and DexFuzz [40] advocate domain aware binary fuzzing to aid VM testing. As we have observed, classfuzz is effective in testing JVMs' startup processes, but much less effective in deep JVM testing. Java* Fuzzer [39] is also syntax directed, aiming to generate tests that can cover more syntax features (class inheritance, complex loop patterns, improved exception throwing patterns, *etc.*). DexFuzz supports random alterations of a seed's control flow, which is close to clrandom in our evaluation. classming differs from these techniques in that it deliberately accepts live, diverse class mutants.

Mutation-based testing. Program mutation, which takes programs as seeds, and then performs mutations, becomes increasingly more important for validating compilers, program execution engines, and virtual machines [10] [40] [41] [42]. Le *et al.* introduce the concept of equivalence modulo inputs (EMI) for testing C compilers [43]. Lidbury *et al.* adapt EMI to fuzz test OpenCL compilers [44]. Sun *et al.* propose Hermes that creates equivalent modulo inputs on live code [27]. Although classming also mutates live bytecode, a mutant and its seed are enforced to be semantically different, rather than equivalent.

Table IV provides a detailed comparison among classming, classfuzz [10], and Hermes [27]. Clearly, the three approaches take their respective strategies and follow their respective processes to generate program mutants. As for the runtime optimization-based JVMs, a set of live, diverse tests can be much more suitable for deep, differential testing them. To the best of our knowledge, classming is the first effort that systematically generates tests for this purpose.

American fuzzy lop (AFL) is a well-known security-oriented fuzzer. It employs compile-time instrumentation and genetic algorithms to discover test cases that trigger new internal states in software binaries [13]. AFL has been extended to fuzz execution engines. Skyfire [45] leverages the vast amount of samples to learn grammar and semantic rules, and then generates seed inputs that can be fed to AFL to fuzz XSLT, XML, JavaScript and rendering engines. Kelinci is an adapation of AFL to fuzz Java programs [46]. It may also be promising to adapt AFL to fuzz JVMs, *e.g.*, by equipping it with domain-specific libraries such as ASM [47] and Soot [14]. Our work on classming still differs as it focuses on manipulating and altering control- and data-flow of seed classes.

The effectiveness of mutation-based testing can be enhanced by the MCMC sampling methods [20] [21] [23] [48] [49]. In terms of JVM testing, both classming and classfuzz adopt the Metropolis-Hastings algorithm to guide their respective mutation processes. The difference is that classfuzz utilizes the algorithm to prioritize its mutators, while classming to effectively produce live, diverse mutants.

VI. Conclusion

Effective, deep JVM testing is an important and challenging task. We have presented LBC mutation and its realization classming to tackle this challenge. The main objective is to generate abundant, diverse, executable classes, and classming achieves this via novel systematic manipulation of the control- and data-flow of live bytecode. Our extensive evaluation results have clearly demonstrated classming's effectiveness and practicality in stress-testing production JVMs and exposing deep JVM differences/defects. Besides continuing our own testing efforts, we plan to make classming publicly available to aid JVM developers in their routine development.

Acknowledgments

Ting Su is the corresponding author. We would like to thank the anonymous ICSE 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). Zhendong Su was supported in part by United States NSF Grants 1528133 and 1618158, and Google and Mozilla Faculty Research awards. Yuting Chen was also partially supported by Shanghai Municipal Commission of Economy and Informatization (No. 201701052).

References

[1] J. E. Smith and R. Nair, *Virtual machines - versatile platforms for systems and processes*. Elsevier, 2005.

[2] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java virtual machine specification - Java SE 9 edition*, 2017. [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se9/html/index.html

[3] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox, "Design of the Java HotSpot client compiler for Java 6," *TACO*, vol. 5, no. 1, pp. 7:1–7:32, 2008.

[4] Oracle, "JDK 9.0.4 general-availability release," 2018. [Online]. Available: http://jdk.java.net/9/

[5] IBM, "Ibm developer kits," 2018. [Online]. Available: https://developer.ibm.com/javasdk/

[6] Azul, "Zing: a better JVM," 2018. [Online]. Available: http://www.azulsystems.com/products/zing/

[7] ——, "Zulu: 100% OpenJDK," 2018. [Online]. Available: http://www.azulsystems.com/products/zulu/

[8] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley, "The jalapeño virtual machine," *IBM Systems Journal*, vol. 39, no. 1, pp. 211–238, 2000.

[9] B. Alpern, S. Augart, S. M. Blackburn, M. A. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. J. Fink, D. Grove, M. Hind, K. S. McKinley, M. F. Mergen, J. E. B. Moss, T. A. Ngo, V. Sarkar, and M. Trapp, "The Jikes research virtual machine project: building an open-source research community," *IBM Systems Journal*, vol. 44, no. 2, pp. 399–418, 2005.

[10] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of JVM implementations," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*, 2016, pp. 85–99.

[11] F. Yellin, "Low level security in Java," *World Wide Web Journal*, vol. 1, no. 1, 1996.

[12] S. N. Freund and J. C. Mitchell, "A type system for the Java bytecode language and verifier," *J. Autom. Reasoning*, vol. 30, no. 3-4, pp. 271–321, 2003.

[13] M. Zalewski, "American fuzzy lop," 2015. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[14] 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*, 1999, p. 13.

[15] E. Bodden, "Inter-procedural data-flow analysis with IFDS/IDE and Soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis (SOAP 2012)*, 2012, pp. 3–8.

[16] Sable Research Group, "Soot: A framework for analyzing and transforming java and android applications," 2012. [Online]. Available: https://sable.github.io/soot/

[17] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White, "An efficient meta-lock for implementing ubiquitous synchronization," in *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99)*, 1999, pp. 207–222.

[18] R. Johnson and K. Pingali, "Dependence-based program analysis," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93)*, 1993, pp. 78–89.

[19] T. Su, K. Wu, W. Miao, G. Pu, J. He, Y. Chen, and Z. Su, "A survey on data-flow testing," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 5:1–5:35, 2017.

[20] D. Huang, J. Tristan, and G. Morrisett, "Compiling Markov chain Monte Carlo algorithms for probabilistic modeling," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*, 2017, pp. 111–125.

[21] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan, "An introduction to MCMC for machine learning," *Machine Learning*, vol. 50, no. 1-2, pp. 5–43, 2003.

[22] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)*, 2013, pp. 305–316.

[23] Y. Chen and Z. Su, "Guided differential testing of certificate validation in SSL/TLS implementations," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*, 2015, pp. 793–804.

[24] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006)*, 2006, pp. 169–190. [Online]. Available: http://www.dacapobench.org/

[25] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Software Eng.*, vol. 37, no. 5, pp. 649–678, 2011.

[26] M. L. Vásquez, G. Bavota, M. Tufano, K. Moran, M. D. Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk, "Enabling mutation testing for android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 233–244.

[27] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*, 2016, pp. 849–863.

[28] Sun Microsystems, Inc, "Tck project planning and development guide," 2003. [Online]. Available: https://jcp.org/aboutJava/communityprocess/ec-public/TCK-docs/ppg.pdf

[29] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko, "SPECjvm2008 performance characterization," in *Proceedings of Computer Performance Evaluation and Benchmarking (SPEC Benchmark Workshop 2009)*, 2009, pp. 17–35.

[30] Standard Performance Evaluation Corporation (SPEC), "SPECjbb2013 design document," 2013. [Online]. Available: http://www.spec.org/jbb2013/docs/designdocument.pdf

[31] R. Pozo and B. Miller, "SciMark 2.0 benchmark," 2004. [Online]. Available: http://math.nist.gov/scimark2/

[32] T. Kalibera, J. Hagelberg, P. Maj, F. Pizlo, B. L. Titzer, and J. Vitek, "A family of real-time java benchmarks," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 14, pp. 1679–1700, 2011.

[33] B. Livshits, "Stanford SecuriBench," 2006. [Online]. Available: http://suif.stanford.edu/~livshits/securibench

[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'99)*, 1999, pp. 1–13.

[35] 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)*, 2003, p. 20.

[36] 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*, 2013, pp. 347–352.

[37] ——, "Automated conformance testing of Java virtual machines," in *Proceedings of the 7th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS 2013)*, 2013, pp. 547–552.

[38] 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)*, 2014, pp. 84–87.

[39] M. R. Haghighat, D. Khukhro, A. Yakovlev, N. Rinskaya, and I. Popov, "Java* fuzzer for android*," 2016. [Online]. Available: https://github.com/AzulSystems/JavaFuzzer

[40] 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)*, 2015, pp. 121–132.

[41] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21th USENIX Security Symposium (SEC 2012)*, 2012, pp. 445–458.

[42] Y. Chen, A. Groce, C. Zhang, W. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*, 2013, pp. 197–208.

[43] V. Le, M. Afshari, and Z. Su, “Compiler validation via equivalence modulo inputs,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’14)*, 2014, pp. 216–226.

[44] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, “Many-core compiler fuzzing,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*, 2015, pp. 65–76.

[45] J. Wang, B. Chen, L. Wei, and Y. Liu, “Skyfire: data-driven seed generation for fuzzing,” in *2017 IEEE Symposium on Security and Privacy (SP 2017)*, 2017, pp. 579–594.

[46] R. Kersten, K. S. Luckow, and C. S. Pasareanu, “Afl-based fuzzing for Java with Kelinci,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*, 2017, pp. 2511–2513.

[47] E. Bruneton, R. Lenglet, and T. Coupaye, “ASM: A code manipulation tool to implement adaptable systems,” 2002.

[48] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, “Guided, stochastic model-based GUI testing of Android apps,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*, 2017, pp. 245–256.

[49] T. Su, “FSMdroid: Guided GUI Testing of Android Apps,” in *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016 (Companion))*, 2016, pp. 689–691.