History-Driven Test Program Synthesis for JVM Testing

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

Zan Wang
College of Intelligence and Computing, Tianjin University
China
wangzan@tju.edu.cn

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

Mengdi Liu
College of Intelligence and Computing, Tianjin University
China
liumengdi@tju.edu.cn

Mingyuan Wu
Southern University of Science and Technology
China
11849319@mail.sustech.edu.cn

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

Lingming Zhang
University of Illinois Urbana-Champaign
United States
lingming@illinois.edu

ABSTRACT

Java Virtual Machine (JVM) provides the runtime environment for Java programs, which allows Java to be "write once, run anywhere". JVM plays a decisive role in the correctness of all Java programs running on it. Therefore, ensuring the correctness and robustness of JVM implementations is essential for Java programs. To date, various techniques have been proposed to expose JVM bugs via generating potential bug-revealing test programs. However, the diversity and effectiveness of test programs generated by existing research are far from enough since they mainly focus on minor syntactic/semantic mutations. In this paper, we propose **JavaTailor**, the first history-driven test program synthesis technique, which synthesizes diverse test programs by weaving the ingredients extracted from JVM historical bug-revealing test programs into seed programs for covering more JVM behaviors/paths. More specifically, JavaTailor first extracts five types of code ingredients from the historical bug-revealing test programs. Then, to synthesize diverse test programs, it iteratively inserts the extracted ingredients into the seed programs and strengthens their interactions via introducing extra data dependencies between them. Finally, JavaTailor employs these synthesized test programs to differentially test JVMs. Our experimental results on popular JVM implementations (i.e., HotSpot and OpenJ9) show that JavaTailor outperforms the state-of-the-art technique in generating more diverse and effective test programs, e.g., test programs generated by JavaTailor can achieve higher JVM code coverage and detect many more unique inconsistencies than the state-of-the-art technique. Furthermore, JavaTailor has detected 10 previously unknown bugs, 6 of which have been confirmed/fixed by developers.

*Junjie Chen is the corresponding author.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510059

CCS CONCEPTS

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

KEYWORDS

Java Virtual Machine, Program Synthesis, JVM Testing, Compiler Testing

ACM Reference Format:
Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-Driven Test Program Synthesis for JVM Testing. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510059

1 INTRODUCTION

Java Virtual Machine (JVM) is the fundamental infrastructure to support the running of Java programs and the programs that are written in other programming languages but can be compiled to Java bytecode [18, 34, 35, 42, 45]. Over the years, many JVMs have been developed by various organizations or companies, such as HotSpot from Oracle [6], OpenJ9 from IBM [9], GIJ from GNU [5], and Zulu from Azul [10]. Although many of them have been elaborately maintained for many years, like other software systems, JVM also contains bugs [44]. Due to its fundamental role, JVM bugs could lead to unexpected behaviors (even disasters in safety-critical domains) of any programs running on top of it. Therefore, it is crucial to ensure JVM's quality.

In recent years, some JVM testing techniques have been proposed to guarantee the quality of JVM, including *classfuzz* [31] and *classming* [30]. These techniques design various mutation operators to generate a large number of Java classfiles (`*.class`) based on real-world classfiles (also called seed classfiles) as test inputs for JVM testing. In this paper, we call JVM's test inputs *test programs* and seed classfiles *seed programs* following the existing work [23]. Specifically, *classfuzz* [31] designs a series of syntactic mutation operators (e.g., changing the modifier or type of a variable). However, its generated test programs are usually invalid, causing that they are rejected at the JVM's startup stage (i.e., loading, linking, and initialization) and thus cannot reach the follow-up verification and execution stages. To get rid of this limitation, the state-of-the-art

technique, i.e., *classming*, was proposed [30], which designs some mutation operators (e.g., inserting `goto` or `return` instructions) to alter the control- and data- flow of seed programs instead of syntactic mutation.

Although these techniques have been demonstrated to be able to detect some new JVM bugs, they still suffer from the effectiveness problem. Specifically, they just aim to generate the test programs with diverse control- and data- flow by accumulating minor mutations, rather than *bug-revealing* test programs. That is, their goal is not directly aligned with the testing goal, leading to spending plenty of time on generating and executing the test programs without the bug-revealing capability and thus hindering their effectiveness. Moreover, the great control- and data- flow diversity of test programs does not mean diverse testing capabilities for the JVM under test. In particular, their minor mutations (e.g., mutating some keywords) actually limit the space for constructing new test programs and thus limit their testing capabilities. For example, in our study (to be presented in Section 4) the state-of-the-art technique (i.e., *classming*) only increases 1.30% JVM line coverage compared with the seed program (i.e., avrora) after executing 3,765 its generated test programs. Therefore, more effective JVM testing techniques are still desirable.

To further improve the effectiveness of JVM testing, in this work we propose a novel technique, called **JavaTailor**, which aims to generate bug-revealing test programs as much as possible in order to approach the ideal testing goal. To achieve this goal, JavaTailor investigates the test programs revealing historical JVM bugs and then extracts bug-revealing ingredients from them to facilitate the generation of new bug-revealing test programs. The key insight lies in that each historically bug-revealing test program contains the ingredients facilitating the detection of the bug, which may involve complicated code logic or cover corner cases. If we combine the ingredients extracted from various historically bug-revealing test programs or put these ingredients into different contexts, it is very likely to generate the test programs that can cover more interested JVM's behaviors/paths, leading to detecting new bugs. With this intuition, we design JavaTailor consisting of three steps: First, JavaTailor extracts the ingredients from our collected historically bug-revealing test programs to form an ingredient pool. In particular, we systematically extract five types of ingredients at the block level, to balance extraction efficiency and effectiveness. Second, JavaTailor generates a new test program by synthesizing a randomly selected ingredient from the pool and a real-world classfile (i.e., a seed program). The seed program is responsible to provide different contexts for the extracted ingredient. The main technical challenge of JavaTailor lies in this step since it is necessary to guarantee the synthesized test program to be valid. Here, JavaTailor designs two strategies (i.e., reusing variables in the seed program and constructing new definitions) to fix the broken syntactic and semantic constraints in the extracted ingredient. In particular, JavaTailor allows synthesizing the seed program with multiple ingredients in an iterative way, which is helpful to combine different bug-revealing ingredients for JVM testing. Third, JavaTailor adopts differential testing to check whether the generated test program can reveal a JVM bug or not.

To evaluate the effectiveness of JavaTailor, we conducted extensive experiments on two popular JVM implementations (i.e., HotSpot and OpenJ9) involving 5 OpenJDK versions, by taking 8 real-world benchmarks as seed programs and collecting 630 historically bug-revealing test programs. Our experimental results demonstrate that JavaTailor is able to detect much more unique inconsistencies than the state-of-the-art JVM testing technique (i.e., *classming* [30]), achieving 792.31% ∼ 1742.86% improvements across all the OpenJDK versions except OpenJDK14 (only JavaTailor detects inconsistencies on this version, and thus we cannot calculate the improvement on it). Also, on the basis of test coverage achieved by the seed programs, JavaTailor is able to further improve much more line coverage, branch coverage, and function coverage than *classming*. In particular, JavaTailor detects 10 unknown bugs in the latest HotSpot and OpenJ9, among which 6 has been confirmed or fixed by developers after submitting them to the corresponding bug repositories. Those results demonstrate the significant effectiveness of JavaTailor.

To sum up, this work makes the following major contributions:

- **Direction.** We open a new direction for JVM testing: while prior work on JVM testing focused on minor syntactic/semantic mutations, our work opens a new dimension for JVM testing via history-driven test program synthesis to cover more diverse JVM paths/behaviors.
- **Technique.** We propose a novel JVM testing technique, called JavaTailor, which aims to generate bug-revealing test programs as much as possible by elaborately utilizing historically bug-revealing test programs.
- **Implementation.** We develop and release a tool to implement JavaTailor [7], including systematically extracting ingredients from historically bug-revealing test programs and synthesizing the ingredients with a given seed program to produce a valid test program.
- **Study.** We conduct an extensive study to evaluate JavaTailor based on popular JVM implementations(i.e., HotSpot and OpenJ9), demonstrating the significant superiority of JavaTailor over the state-of-the-art JVM testing technique. In particular, JavaTailor has detected 10 unknown bugs, 6 of which have been confirmed or fixed by developers.

2 MOTIVATION AND CHALLENGES

Here, we use an example to illustrate the motivation of JavaTailor and its major challenges.

Figure 1a shows a test program generated by JavaTailor, which detects an unknown OpenJ9 bug. This bug is caused due to missing `null` checks for the parameters in `MemoryNotificationInfo` in the OpenJ9 implementation. `MemoryNotificationInfo` is an internal class under `java.lang.management` package, which is used to notify when the memory usage exceeds a threshold. The notified information is vital for debugging when error occurs. When running this test program with a `null` parameter (i.e., `usage`) on OpenJ9, it is executed normally without any exception. However, when running the same test program on HotSpot, a `NullPointerException` is thrown since its implementation contains `null` checks for the parameters in `MemoryNotificationInfo`. The different behavior exhibited by them indicates the existence of a bug in at least one of them. Through our manual investigation and submitting a bug report to the bug repository of OpenJ9, the bug was confirmed

```
1  public static void main(String[] args) {
2    String str = "anystring";
3
4    String name = str;
5    MemoryUsage usage = null;
6    long count = 999;          //generate by random
7    MemoryNotificationInfo mn;
8    mn = new MemoryNotificationInfo(name, usage, count);
9    count = mn.getCount();
10
11   System.out.println(str);
12 }
```

(a) Test program

```
1  public static int run(String[] argv, PrintStream out) {
2    MemoryUsage mu = new MemoryUsage(1, 2, 3, 4);
3    ...
4    // Check negative count
5    mn = new MemoryNotificationInfo("poolName", mu, -1);
6    count = mn.getCount();
7    if (count != -1) {
8       out.println("FAILURE 2.");
9       out.println("Wrong count: " + count + ", expected: -1");
10      testFailed = true;
11   }
12   ...
13 }
```

(b) Historical test program

Figure 1: Motivating example

```
   public MemoryNotificationInfo(String poolName,
                                 MemoryUsage usage, long count) {
-     super();
+     if (poolName == null) {
+        /*[MSG "K0D02", "Null poolName"]*/
+        throw new NullPointerException(***.getString("K0D02"));
+     }
+
+     if (usage == null) {
+        /*[MSG "K0D03", "Null usage"]*/
+        throw new NullPointerException(***.getString("K0D03"));
+     }
      this.poolName = poolName;
      this.usage = usage;
      this.count = count;
   }
```

Figure 2: OpenJ9 Bug#12552

and fixed by OpenJ9's developers (Bug ID: 12552 [2]). The patch provided by them is shown in Figure 2, in which `null` checks have been added for `MemoryNotificationInfo`'s parameters.

This bug-revealing test program is generated by synthesizing the ingredient extracted from a historical bug-revealing test program (as shown in Figure 1b) and an arbitrarily selected seed program (as shown in Figure 1a without the code marked in red). Please note that the historical test program cannot trigger the above detected bug since no parameters in `MemoryNotificationInfo` are set to `null`. Specifically, we extract Lines 5-6 from the historical test program as the ingredient and then insert it to the seed program. As shown in Figure 1a, we insert the ingredient at Lines 4-9 to produce a test program. Here, directly inserting the ingredient to the seed program cannot make the synthesized test program valid due to lack of definition of some variables (such as `mn` and the parameters of `MemoryNotificationInfo`), and we have to conduct extra operations to make it valid.

To better integrate the ingredient and seed program for the triggering of interesting/corner-case interactions between them, we prefer to replace the undefined variables in the ingredient with the existing variables satisfying type compatibility in the seed program. Thus, we assign `str` defined at Line 2 to the first parameter of `MemoryNotificationInfo` (as shown at Line 4). If the seed program also does not contain such type-compatible variables, we have to generate definitions for the corresponding variables. For example, the second parameter of `MemoryNotificationInfo` belongs to the type of `MemoryUsage` and thus we generate its definition as shown at Line 5. Since the second parameter is initialized to be `null`, the synthesized test program is able to trigger the bug.

We further analyzed whether existing JVM testing techniques can detect this bug. Regardless of *classming* or *classfuzz*, they just conduct minor mutations (e.g., changing the modifier of a variable or inserting the `goto` keyword) on the seed program in an iterative way, which aims to change the data- and control- flow inside the seed program. If the seed program does not contain `MemoryNotificationInfo` like the one used in the example (i.e., the seed program contains only Lines 2 and 11 as shown in Figure 1a), these techniques cannot generate test programs revealing the bug no matter how to change its data- and control- flow. As presented above, however, synthesizing the ingredients from historical test programs is more likely to introduce bug-revealing program features, indicating the promising direction of mining the ingredients accumulated in a large number of historical bug-revealing test programs for constructing better JVM test programs.

While it is a promising direction, synthesizing new bug-revealing test programs based on historically bug-revealing test programs is not trivial, which suffers from two major challenges:

Challenge 1: *How to measure and extract ingredients in historically bug-revealing test programs?* A test program tends to contain various language structures, which can be represented at different granularities (such as variables, blocks, or files). If we measure and extract ingredients at a very fine-grained granularity (e.g., variable granularity), the process of ingredient extraction could become costly and the whole syntactic or semantic features relevant to bug detection may be damaged. If the granularity is too coarse (e.g., file granularity), the interaction between the extracted ingredients and the seed program could be weak, which affects the integration of the ingredients with new contexts and thus impairs the testing performance of synthesized test programs. Therefore, measuring and extracting ingredients at an appropriate granularity is important and non-trivial.

Challenge 2: *How to guarantee that a synthesized test program is valid?* A test program usually involves various syntactic and semantic constraints. Violating them can make it become invalid, and an invalid test program will be rejected by the JVM under test, thus impairing the testing performance. When synthesizing the ingredients extracted from one test program with another test program, it is scarcely possible to produce a valid test program by directly combining them since they tend to involve very different syntactic and semantic constraints. Therefore, it is very important but challenging to make the different constraints from the two test

programs compatible during the synthesis process, so that a valid test program can be produced.

3 APPROACH

In this paper, we propose a novel JVM testing technique, called **JavaTailor**, which aims to generate new bug-revealing test programs by synthesizing the code ingredients extracted from historically bug-revealing test programs with given seed programs. The key insight behind JavaTailor is that 1) existing test programs revealing historical bugs contain the code ingredients facilitating the detection of bugs, which tend to contain more complicated code logic or cover various corner cases, and 2) combining various such code ingredients and/or putting them into different code contexts can potentially cover even more interesting JVM behaviors/paths important for new JVM bug detection.

Figure 3 shows the overview of JavaTailor, which consists of three stages. First, JavaTailor extracts the ingredients from the collected test programs revealing historical bugs to construct an ingredient pool (Section 3.1). More specifically, we systematically design various types of code ingredients at the block granularity in JavaTailor to balance both effectiveness and efficiency. Second, JavaTailor synthesizes a randomly selected ingredient from the ingredient pool with a given seed program to generate a valid test program (Section 3.2). Meanwhile, JavaTailor fixes the broken syntactic and semantic constraints in the code ingredient (i.e., missing variables' definitions) by either utilizing the type-compatible variables in the seed program or automatically constructing the missing definitions. Third, JavaTailor executes the synthesized test program for JVM testing (Section 3.3). It adopts differential testing based on different JVM implementations (e.g., HotSpot and OpenJ9) to check whether the test program reveals a bug or not. If there is no inconsistencies identified by the synthesized test program, the test program will be put into the pool of seed programs for further combining with more ingredients.

3.1 Ingredient Extraction

As discussed in Section 2, the effectiveness of extracted ingredients could be affected by its extraction granularity. In JavaTailor, we adopt the block granularity as the trade-off between extraction efficiency and effectiveness. Specifically, we systematically deign five types of blocks for ingredient extraction as follows. In particular, our implementation for JavaTailor is based on Soot [47], a widely-used tool for analyzing Java classfiles, and our five types of blocks can cover all the types of instructions supported by Soot. Based on Soot, JavaTailor extracts ingredients from the Java classfile (i.e., Jimple code transformed by Soot) level rather than the source code level, which can acquire the following benefits: 1) There are a number of operations on Jimple code supported by Soot, facilitating the implementation of JavaTailor; 2) It facilitates to generate more test programs with richer semantics by getting rid of the constraints from front-end compilers (e.g., javac).

- **Sequential Ingredient (SEQ)**. A SEQ refers to a block containing a sequence of instructions without any branches.
- **If Ingredient (IF)**: A IF could have several branches (i.e., `if`, `else if`, and `else`), and it includes the conditions of all the branches and the corresponding bodies.
- **Loop Ingredient (LOOP)**: A LOOP could be `while`, `do-while`, or `for` LOOP. It includes the loop condition and the corresponding body.
- **Switch Ingredient (SWITCH)**: A SWITCH includes the condition and all the cases. In Soot, it contains both `lookupswitch` and `tableswitch`.
- **Try-Catch Ingredient (TRAP)**: It includes the `try` body and the statements used for handling the caught exception.

To implement the extraction of ingredients, JavaTailor transforms a historical bug-revealing test program into a Control-Flow Graph (CFG) based on the Jimple code obtained after Soot's processing. In a CFG, a node refers to a basic block including one or more instructions, and an edge represents the code logic between two basic blocks. An ingredient in JavaTailor includes one or more basic blocks in a CFG.

Based on the CFG of a test program, JavaTailor first identifies the starting points (i.e., basic blocks) of the latter four types of ingredients according to the types of instructions included in each basic block. For example, if a basic block contains a `switch` instruction, it can be regarded as the starting point of a SWITCH ingredient. Then, for each starting point of an ingredient, JavaTailor searches for its dependent basic blocks to form a whole ingredient. For example, after identifying a basic block including a `switch` instruction, JavaTailor searches for the basic blocks for all the cases of this `switch` condition, and finally all these basic blocks form a SWITCH ingredient. There is a special case in implementations and we further illustrate it in detail: If a basic block contains an `if` instruction, it is hard to determine whether it is the starting point of an IF ingredient. This is because in Soot, loops are represented as the combination of `if` and `goto` instructions. Thus, we need to check whether its successor basic blocks contain a `goto` instruction. If a `goto` instruction is found and its target is the starting point, this basic block is actually the starting point of a LOOP ingredient; otherwise, it is the starting point of an IF ingredient. For a basic block that cannot be identified as the starting point of any of the latter four types of ingredients, JavaTailor treats the basic block as a SEQ ingredient.

To further illustrate the extraction process, we take an example shown in Figure 4, which is a CFG of nested `for-if`. For each basic block in the CFG, JavaTailor checks whether it is the starting point of one of the ingredients. For example, for the block labeled as 2 (also called block 2), it contains an `if` instruction, indicating that it may be the start point of an IF ingredient or a LOOP ingredient. To figure out its type, JavaTailor recursively gets all its successor blocks, and finds that there is a `goto` instruction that points to block 2 in block 5, indicating that this is a LOOP ingredient. It corresponds to the part labeled as FOR in this figure. For block 3, there is no `goto` instruction pointing to it in all its successor blocks, and thus it is an If Ingredient, corresponding to the part labeled as IF in this figure. When the extraction process is completed, JavaTailor can extract 4 ingredients in this example, as shown at the bottom of Figure 4. Note that the start and end blocks of CFG are filtered here, due to its simple code logic.

Through extracting the five types of ingredients from all the collected historical bug-revealing test programs, an ingredient pool can be built by JavaTailor.

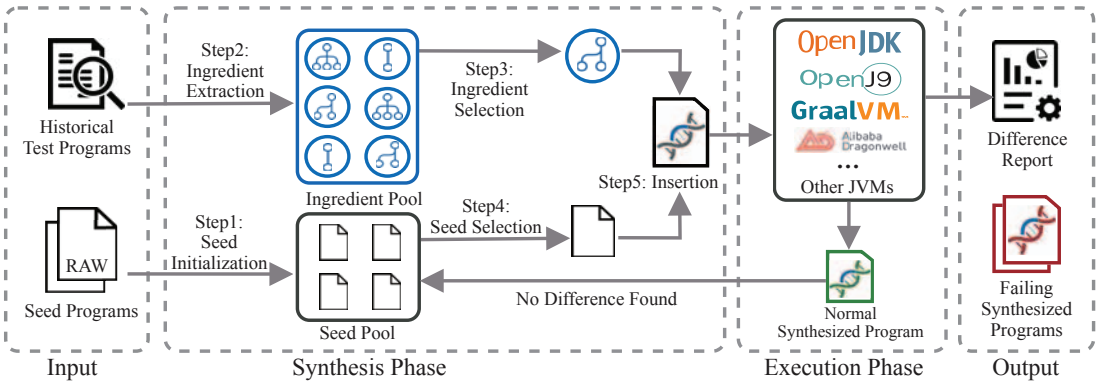

Figure 3: Overview of JavaTailor

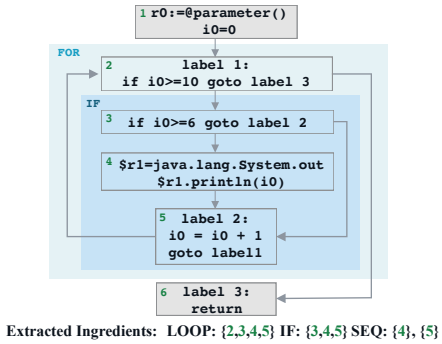

Figure 4: Control flow graph of nested For-If

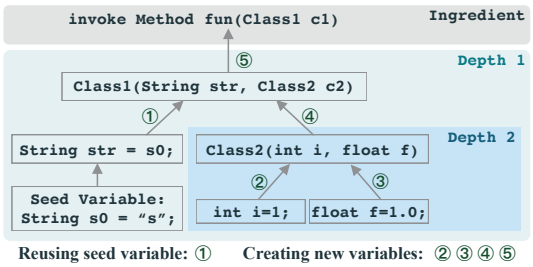

Figure 5: Fix broken constraints

3.2 Test Program Synthesis

To generate a new test program, JavaTailor randomly selects an ingredient from the ingredient pool as well as a seed program, and then synthesizes the ingredient with the seed program. Also, JavaTailor randomly selects a program point in the seed program for inserting the extracted ingredient (we call it *synthesis point* in this paper). Such random operations are helpful to generate *diverse* synthesized test programs. As discussed in Section 2, it is challenging to ensure that the synthesized test program is valid due to breaking the original syntactic and semantic constraints of the ingredient (i.e., missing variable definitions). Therefore, to obtain a valid synthesized test program, JavaTailor has to fix those broken constraints during the synthesis process. To achieve this goal, JavaTailor has two strategies, i.e., reusing variables in the seed program and constructing new definitions.

Reusing variables in the seed program. For a variable missing its definition in the ingredient, JavaTailor prefers to find whether there is a variable in the seed program that can be used to replace the variable in the ingredient. In this way, the interaction between the ingredient and the seed program can be stronger, enabling the new context to produce larger impact on the historical bug-revealing ingredient, which in turn is more likely to trigger different JVM behaviors and reveal new bugs. Specifically, JavaTailor searches for the *type-compatible* variables in the seed program with the variable missing its definition in the ingredient from the code before the synthesis point. If such a variable is found, JavaTailor then replaces the ingredient's variable with the identified type-compatible variable, in order to recover the broken constraints.

Constructing new definitions. Not all the variables missing definitions in the ingredient can find such type-compatible variables in the seed program, especially when the variable type is an `Object` type. At this time, JavaTailor has to construct definitions for those variables in order to fix the broken constraints. Algorithm 1 formally illustrates the definition construction process. Regarding the primitive types, JavaTailor directly constructs the corresponding types of variables with a random initialization at Line 5. Otherwise, if a variable's type is an `Object` type, we can invoke the corresponding constructor to define the variable at Line 9, but it is also very likely to come across the parameters with other `Object` types in the constructor as shown in Lines 10-15. Then, JavaTailor is also required to define these parameters by invoking their constructors at Line 14. That is, this process is recursive until all the required parameters are primitive types. In particular, to avoid costly or even endless recursion, JavaTailor sets a maximal recursion depth (denoted as $\mathcal{D}$) as the terminating condition in Lines 2-3. If the recursion depth exceeds $\mathcal{D}$, JavaTailor directly initializes the corresponding parameters as `null`.

Figure 5 shows a simple example to further illustrate the process of fixing broken constraints in JavaTailor, where the selected ingredient is a function call with a parameter `c1` of the `Class1` type. To create the definition of `c1`, JavaTailor first needs to create variables for the parameters of its constructor (i.e., `String str`

Algorithm 1: Constructing new definitions

```
Input: type: the missing type, depth: current iteration depth
Output: v: newly generated variable
Function CreateVarWithType(type, depth):
    if depth > D then
        return null;
    if type(t) is primitive type or String then
        v ← randomly create variable with type(t);
    else
        cs ← identify all constructors of type;
        if cs contains non-parameter constructor then
            v ← create variable with non-parameter constructor;
        else
            c ← randomly select a constructor in cs;
            ts ← identify all parameter types of c;
            for each t ∈ ts do
                p_t ← p_t ∪ CreateVarWithType(type(t), depth + 1);
            v ← create variable with c and p_t;
    return v;
```

and `Class2 c2`). Since there is a `String` type variable defined in seed program, we reuse it at ①. However, there is no variable of type `Class2` defined in the seed program, JavaTailor needs to recursively create variables of `Class2` and its parameters of types `int` and `float` in the next recursion. Since the missing dependent variables of `Class2` are all primitive types, we randomly create a value for them at ② and ③. Then, the variable of `Class2` can be created at ④ and finally create the definition of `Class1 c1` at ⑤.

After fixing those broken constraints, JavaTailor inserts the processed ingredient into the synthesis point. Please note that JavaTailor replaces the `return` instructions with `goto` instructions in order to avoid terminating the synthesized test program prematurely. Also, it is necessary to assign a `label` for each `goto` instruction. To boost the interaction between the ingredient and the seed program, JavaTailor inserts the `label` to the code belonging to the seed program.

3.3 Synthesized Program Execution

After generating a new test program via synthesis, JavaTailor then executes it and checks whether it reveals a JVM bug or not. Here, JavaTailor adopts differential testing as the test oracle, which compares the outputs of different JVM implementations (such as HotSpot and OpenJ9) with regard to this test program. Since a test program may produce a large amount of outputs, which may include non-deterministic outputs (such as time-related outputs), it may incur the inaccuracy when determining an inconsistency. Moreover, faced with the same exception, different JVM implementations may also produce different stack traces, further aggravating the difficulty of determining an inconsistency.

JavaTailor relieves this challenge from the following three scenarios: 1) If one JVM terminates normally while another JVM crashes, JavaTailor regards it as an inconsistency without doubts; 2) If both JVM implementations crash during the execution of the same synthesized test program, JavaTailor extracts the exception messages from the produced stack traces by employing regular expressions (such as identifying the lines including the keywords of *Exception*, *Error* and *Failure*), in order to reduce the influence of different styles of stack traces produced by different JVM implementations. Then, if the extracted exception messages are different, JavaTailor regards it as an inconsistency. 3) If both JVM terminates normally, the outputs are also produced by the synthesized test program. To reduce the noise of determining an inconsistency, JavaTailor first filters out some non-deterministic messages by employing regular expressions (such as including the keywords of *time*, *random* and *thread*, and some common time formats), and then filters out the messages produced by the third-party libraries (such as JUnit and Log4J) used by the test program. Then, if the remaining outputs are still different, we regards it as an inconsistency. Regarding the latter two scenarios, we further manually check whether the identified inconsistency is a really bug or a false positive before reporting it to the JVM's developers. In particular, if an inconsistency is a false positive, we further design a rule with regard to it and then incorporate it in JavaTailor to further boost the accuracy of determining an inconsistency.

Please note that if there is no inconsistency identified by a synthesized test program, JavaTailor puts it into the pool of seed programs. In this way, it can be used as a seed program for the following synthesis, and thus a test program combining multiple ingredients from different historical bug-revealing test programs could be generated, which may be more helpful to reveal new JVM bugs.

4 EVALUATION

In the study, we aim to address the following research questions:

- **RQ1**: How does JavaTailor perform in detecting JVM inconsistencies?
- **RQ2**: Can JavaTailor achieve higher JVM coverage?
- **RQ3**: Are the ingredients extracted from historically bug-revealing test programs more effective than existing mutation operators for JVM testing?
- **RQ4**: Can JavaTailor detect previously unknown bugs in the latest JVM implementations?

4.1 Evaluation Settings

Subjects. Following the existing work [30], we adopted two popular JVMs, i.e., HotSpot [6] and OpenJ9 [9], as subjects. Table 1 shows the subjects used in our study. We did not use OpenJDK9 and OpenJDK10 since they are no longer maintained in OpenJ9. We can find that for each OpenJDK version, we used one relatively old build and one latest build for each JVM. Here, to investigate the effectiveness of JavaTailor based on more significant results in statistics, we used these relatively old JVM builds as the subjects under test since they tend to contain more bugs. We call an experiment based on the relatively old HotSpot build and the relatively old OpenJ9 build of one OpenJDK version *a differential-testing experiment*. In total, we have five differential-testing experiments due to evaluating on five OpenJDK versions. Regarding these latest JVM builds, they are used to determine whether an inconsistency detected by JavaTailor on the relatively old builds in a differential-testing experiment is a real one by checking whether the inconsistency has been fixed by the latest JVM builds (more details about it will be presented in the

Table 1: Studied JVM implementations for differential testing

OpenJDK Version	JVM Implementation	JVM Version
OpenJDK8	HotSpot	build 25.0-b70
		build 25.292-b10
	OpenJ9	build openj9-0.8.0
		build openj9-0.26.0
OpenJDK11	HotSpot	build 11+2
		build 11.0.11+9
	OpenJ9	build openj9-0.12.0
		build openj9-0.26.0
OpenJDK12	HotSpot	build 12+33
		build 12.0.2+10
	OpenJ9	build openj9-0.13.0
		build openj9-0.15.1
OpenJDK13	HotSpot	build 13+33
		build 13.0.2+8
	OpenJ9	build openj9-0.16.0
		build openj9-0.18.0
OpenJDK14	HotSpot	build 14+36-1461
		build 14.0.2+12
	OpenJ9	build openj9-0.20.0
		build openj9-0.21.0

part of *evaluation metrics*). In particular, we also applied JavaTailor to test the latest JVM builds to investigate whether it can detect previously unknown JVM bugs.

Historical Bug-Revealing Test Programs. We collected the test programs revealing historical bugs from the HotSpot test suite. This is because it well integrates the bug-revealing test programs from its bug repositories, and each bug-revealing test program in its test suite is equipped with the corresponding bug description, which is convenient for us to distinguish whether a test program in the test suite is bug-revealing or just a normal test. In particular, we removed the test programs that can reveal bugs directly on the subjects under test in order to clearly investigate the effectiveness of JavaTailor. Moreover, we filtered out the test programs that cannot run successfully in our experimental environment. Finally, we collected 630 bug-revealing test programs in total. Based on them, JavaTailor extracts a large number of ingredients (i.e., 33,002), including 17,716 SEQ ingredients , 8,914 IF ingredients, 6,122 LOOP ingredients, 236 TRAP ingredients and 14 SWITCH ingredients.

Seed Programs. We collected the benchmarks that were used in the existing JVM testing study [30] and the test programs from the test suites of HotSpot and OpenJ9 as seed programs. Table 2 shows the basic information of our used seed programs. The former six benchmarks are selected from the existing study [12], and only one classfile (the one including `main` function) in each of them is used as the seed program following the study [30]. There are some other benchmarks used in the existing study [12, 30], but they cannot run successfully in our experimental environment due to their old/outdated versions. The fourth column shows the number of Jimple instructions in the seed programs for each benchmark.

Table 2: Benchmarks description

ID	Project	#size	#inst	#iter
P1	avrora	1	302	5000
P2	eclipse	1	2061	20000
P3	pmd	1	840	10000
P4	jython	1	377	6000
P5	fop	1	187	3000
P6	sunflow	1	308	4000
P7	HotSpot-tests	630	136823	3 days
P8	Openj9-tests	1616	273710	3 days

Besides, we constructed two interesting scenarios by using the test programs from the test suite of HotSpot and OpenJ9 as seed programs. First, we used the above 630 historically bug-revealing test programs from HotSpot as seed programs, which is helpful to investigate the power of integrating various historical bug-revealing test programs. Second, we also collected the test programs from the test suite of OpenJ9 as seed programs, which is interesting to investigate whether the ingredients from the test programs in one JVM (i.e., HotSpot) can augment the effectiveness of the test programs in the other JVM (i.e., OpenJ9). Similarly, we discarded the test programs that can reveal bugs on our used subjects or cannot run successfully in our experimental environment. In total, we obtained 1,616 test programs from the test suite of OpenJ9 as seed programs.

Compared Approaches. In the study, we compared JavaTailor with the state-of-the-art JVM testing approach, i.e., *classming* [30]. *classming* designs several mutation operators to *minorly* modify a seed program, aiming to alter the control- and data- flow of the seed program, which includes the insertion of five keywords: `goto`, `return`, `throw`, `lookupswitch`, and `tableswitch`. Different from the testing process of JavaTailor (i.e., for generating each test program, it randomly selects an ingredient and a seed program for synthesis), *classming* incorporates the MCMC (Markov Chain Monte Carlo) algorithm to guide the selection of mutation operators in order to iteratively mutate a given test program for generating a series of mutated test programs.

To further evaluate the contribution of our ingredient synthesis method, we mitigate the difference of the testing process between JavaTailor and *classming*. Specifically, we constructed a variant of JavaTailor by replacing our ingredient synthesis method with the minor mutation operators proposed in *classming*, which is called *Javaming*. That is, for generating each test program, *Javaming randomly* selects a mutation operator and a seed program, and then applies the mutation operator to the seed program to produce a new test program. If it does not reveal a JVM bug, this test program will be placed into the pool of seed programs for future selections.

Implementation and Environment. We implemented JavaTailor based on OpenJDK8 (a popular OpenJDK version) and Soot (a mature program analysis tool for Java classfiles that has been widely used in the existing work [30, 31, 41]). Regarding using Soot in implementing JavaTailor, it definitely can be replaced with other libraries, but we chose Soot for a fair comparison with the state-of-the-art classming (which is also implemented on Soot). In JavaTailor, the maximal recursion depth (i.e., $\mathcal{D}$) of constructing

new definitions is set to 5. Since the implementation for *classming* is not available, we carefully re-implemented it based on the description in the existing work [30]. All the settings are consistent with the existing work [30]. In particular, we ran our re-implementation of *classming* according to the study design of its original work [30], and indeed obtained very similar results, which demonstrates the validity of our re-implementation.

All our experiments are conducted on a sever with two dodecacore CPUs Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz and 251GB RAM, running Ubuntu 18.04.4 LTS (64 bit). The implementation of JavaTailor and our datasets can be found at our project homepage [7] for future usage and replication.

Evaluation Metrics. We considered three metrics to measure the effectiveness of JavaTailor. The first one is *the number of unique inconsistencies*. To answer RQ1, we applied each JVM testing approach to each pair of JVM builds (i.e., the pair of relatively old JVM builds of HotSpot and OpenJ9) in each differential-testing experiments. During the given testing period, each approach may detect a number of inconsistencies between each pair of JVM builds. As presented in Section 3.3, an inconsistency may be a real bug or a false positive, and we further ran each approach on the corresponding pair of the latest builds of HotSpot and OpenJ9 in order to check whether the inconsistency still exists or not. If the inconsistency disappears, we regarded it as a bug and this bug has been fixed in the latest builds; Otherwise, we further manually investigated it to obtain the conclusion of the inconsistency. Also, some inconsistencies may be duplicated due to the same bug, and thus we further de-duplicated them according to the crash messages since most of inconsistencies involved the crashes of at least one JVM implementation in each pair in our study. We call the number of inconsistencies after de-duplication *the number of unique inconsistencies*. Relying on crash messages may not achieve perfect de-duplication, but it is the most widely-used *automatic* method to identify unique failures in the existing work [29]. Indeed, it is a potential threat and we will design more accurate metrics in the future.

Since we also applied JavaTailor to test the latest JVM builds in RQ4, we cannot use the above method to *automatically* determine whether each detected inconsistency is a real bug or a false positive. Here, we manually investigated them, and then created and submitted a bug report to the corresponding bug repository for each potential bug after our manual investigation. Then, we can obtain *the number of detected unknown bugs* according to developers' feedback, which is the second metric in our study.

Furthermore, we further measured *the test coverage of JVM* (that is the third metric in our study) achieved by each approach, in order to deeply understand the effectiveness difference between JavaTailor and *classming*. Here, we measured the widely-used line coverage, branch coverage, and function coverage, respectively.

4.2 Process

To answer RQ1, in each differential-testing experiment, we ran each approach for the same testing period on each benchmark. For the former six benchmarks, the existing work proposing *classming* provides the number of iterations for each of them. Here, we kept the same setting for *classming* to show its expected effectiveness and recorded the testing time spent on completing the corresponding iterations on each benchmark, then ran JavaTailor for the same testing time on the corresponding benchmark for fair comparison with *classming*. For the latter two benchmarks, we ran each approach for three days respectively. Since *classming* is an iterative process on a seed program, it is required to set the number of iterations on each seed program in the two benchmarks. According to the former six benchmarks, we can obtain that one instruction requires 14 iterations on average. Thus, for the latter two benchmarks, after selecting a seed program, we set the number of iterations on it to $14 * $ *the number of instructions of the seed program* for *classming*. The testing process of each approach terminates after running for three days for fair comparison. To answer RQ3, we ran *Javaming* following the same setting of RQ1.

To answer RQ2, we took HotSpot *build 11-internal+0* for OpenJDK11 as the representative, for test coverage collection. The running process of each approach is consistent with the setting of RQ1. To collect the coverage of HotSpot, we compiled it with the flag `–enable-native-coverage`, and then adopts Gcov [4] and Lcov [8] to collect and analyze the line coverage, branch coverage, and function coverage achieved by each approach. In particular, we consider all the source code irrelevant to the underlying platforms in HotSpot for coverage collection. In total, there are *331,978* lines of code, *199,173* branches, and *95,351* functions.

To answer RQ4, we applied JavaTailor to the latest JVM builds, and ran it on each differential-testing experiment for a longer testing time (i.e., five days). Since manually analyzing and reporting each unknown bug is time-consuming, especially the process of reducing a bug-revealing test program into a small but still bug-revealing one, we chose OpenJDK8 and OpenJDK11 as the representatives for testing in this experiment.

4.3 Results and Analysis

4.3.1 RQ1: Effectiveness of JavaTailor. Table 3 shows the comparison results among JavaTailor, *classming*, and *Javaming* in terms of the number of detected unique inconsistencies.

By comparing JavaTailor and *classming*, we found that JavaTailor is able to detect much more unique inconsistencies than *classming* for each differential-testing experiment on each benchmark. By taking the differential-testing experiment for OpenJDK8 as an example, *classming* detects unique inconsistencies on four benchmarks and the total number of unique inconsistencies is 35, while JavaTailor detects unique inconsistencies on all the eight benchmarks and the total number of unique inconsistencies is up to 377. The improvements of JavaTailor over *classming* range from 792.31% to 1742.86% across all the differential-testing experiments except OpenJDK14 (only JavaTailor detects inconsistencies on this version and thus we cannot calculate the improvement on it) in terms of the total number of unique inconsistencies on all the benchmarks. The results demonstrate the significant superiority of our proposed approach JavaTailor over the state-of-the-art approach *classming*.

We further analyzed the reason why JavaTailor significantly outperforms the state-of-the-art approach *classming*. The latter focuses on exploring various control- and data- flow of the seed program based on the ingredients itself in an iterative way, which actually limits the space of constructing test programs. Different from

Table 3: JVM inconsistencies detection effectiveness comparison

ID	OpenJDK8			OpenJDK11			OpenJDK12			OpenJDK13			OpenJDK14		
	C.M.	J.T.	J.M.	C.M.	J.T.	J.M.	C.M.	J.T.	J.M.	C.M.	J.T.	J.M.	C.M.	J.T.	J.M.
P1	0	8	0	0	3	0	0	3	0	0	1	0	0	0	0
P2	0	14	2	0	8	0	0	2	0	0	6	0	0	0	0
P3	0	9	2	0	6	0	0	1	0	0	3	0	0	1	0
P4	0	8	0	0	2	0	1	2	0	1	1	0	0	0	0
P5	3	12	0	4	17	4	4	18	0	0	1	0	0	0	0
P6	2	20	0	0	8	0	0	5	0	1	4	0	0	1	0
P7	4	143	7	1	122	3	2	48	3	2	30	0	0	16	0
P8	26	163	10	9	92	8	10	95	8	9	70	4	0	70	0
Total	**35**	**377**	**21**	**14**	**258**	**15**	**17**	**174**	**11**	**13**	**116**	**4**	**0**	**88**	**0**

C.M. and J.T. are the abbreviations of *classming* and JavaTailor respectively

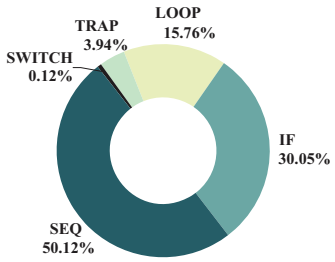

Figure 6: Inconsistency distribution by ingredient types

Table 4: Confirmed/Fixed unknown bugs

Bug ID	JVM	Affected Versions	Status
Bug#12819	OpenJ9	OpenJDK 8, 11, 16	Fixed
Bug#12992	OpenJ9	OpenJDK 8, 11, 16	Fixed
Bug#12552	OpenJ9	OpenJDK 8, 9, 10, 11	Fixed
Bug#12815	OpenJ9	OpenJDK 8, 11, 16	Confirmed
Bug#13242	OpenJ9	OpenJDK 8, 11, 16	Confirmed
JDK-8271457	HotSpot	OpenJDK 9, 11, 17	Confirmed

classming, JavaTailor incorporates various bug-revealing ingredients from historical test programs to generate new test programs, which not only can construct the test programs with diverse (even bug-revealing) control- and data- flow but also enlarges the test program space to increase the chance of producing bug-revealing test programs. Besides, according to the results on the benchmark of HotSpot's test programs, JavaTailor detects much more unique inconsistencies than *classming*, demonstrating that combining various ingredients from different historically bug-revealing test programs is more effective than just individually exploring each historically bug-revealing test program. According to the results on the benchmark of OpenJ9's test programs, JavaTailor also detects a large number of unique inconsistencies, showing that the testing capability of one JVM's test suite can be augmented by another JVM's test suite.

We also investigated whether each type of ingredients can help detect some unique inconsistencies, whose results are shown in Figure 6. This figure presents the percentage of each type of ingredients resulting in the detection of unique inconsistencies. Here, we integrated the results of all the differential-testing experiments. We found that every type of ingredients are able to detect unique inconsistencies. In particular, there are a small number of SWITCH ingredients in our dataset (i.e., 14), but they also revealed unique inconsistencies. That demonstrates the contribution of each type of ingredients. As expected, for each type of ingredients, the number of detected unique inconsistencies is strongly correlated to the number of the type of ingredients in our dataset. For example, we extracted the largest number of SEQ ingredients and indeed it detected the largest number of unique inconsistencies.

4.3.2 RQ2: JVM's Coverage Comparison. We compared JavaTailor and *classming* in terms of JVM's coverage (including line coverage, branch coverage, and function coverage) achieved by them respectively, in order to further explain why JavaTailor performs better than *classming*. Figure 7 shows the coverage comparison results, where the gray lines present the coverage achieved by the seed programs, and the green and yellow lines present the coverage achieved by JavaTailor and *classming* on the basis of the seed programs respectively. We found that regardless of line coverage, branch coverage, or function coverage, the improved coverage by *classming* over the seed programs is very small on each benchmark, indicating that just altering control- and data- flow of the seed programs based on their own ingredients is hard to bring large JVM coverage increments. Regarding JavaTailor, its improved JVM coverage over both *classming* and the seed programs is obvious on all the benchmarks (except P7). The results demonstrate that incorporating more ingredients from other test programs into the seed programs are more helpful to improve JVM coverage, resulting in the detection of more unique inconsistencies than *classming*. The reason for P7 is that both the ingredient pool and the seed programs are from the same test programs in the HotSpot's test suite, leading to the small coverage increments. However, we can observe that the improvement in terms of branch coverage on P7 is larger than that in terms of line coverage and function coverage, indicating that combining various ingredients from different bug-revealing test programs facilitates to coverage more interesting JVM branches/paths and thus can reveal more unique inconsistencies.

4.3.3 RQ3: Comparison between JavaTailor and Javaming. We further compared our ingredient synthesis method and existing minor mutation operators by mitigating the influence of the testing process by comparing JavaTailor and *Javaming*. The columns "J.T." and "J.M." in Table 3 show the comparison results. We obtained the similar conclusions with RQ1, i.e., JavaTailor detects much more unique inconsistencies than *Javaming*. The improvements of JavaTailor over *Javaming* range from 1482.82% to 2800.00% across all the differential-testing experiments except OpenJDK14 (only JavaTailor detect inconsistencies on this version) in terms of the total number of unique inconsistencies on all the benchmarks, demonstrating the significant superiority of our ingredient synthesis method over the existing elaborately designed minor mutation operators. In addition, *classming* performs better than *Javaming* in general, demonstrating

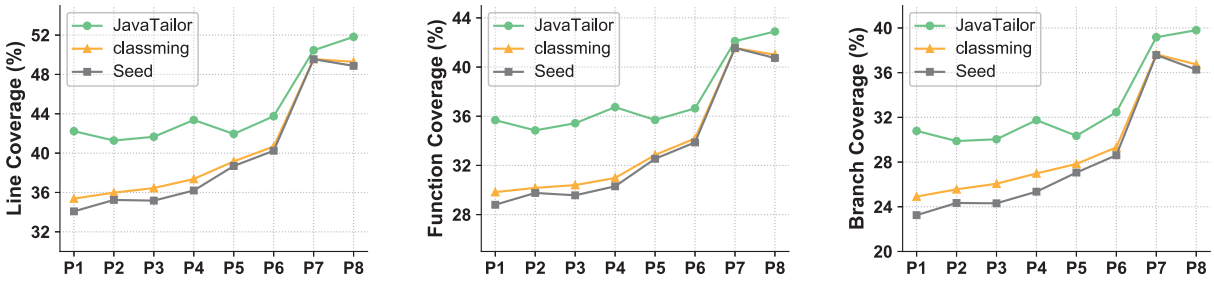

Figure 7: JVM code coverage comparison

```
1 public static Method main:"([Ljava/lang/String;)V"
2   stack 5 locals 6
3 {
4       new   class java/util/ArrayList;
5       dup;
6       invokespecial    Method java/util/ArrayList."<init>":"()V";
7       astore_0;                                  // stores ArrayList
8       ...
9  L13: stack_frame_type full;
10      aload_0;                                   // loads ArrayList
11      checkcast    class java/util/List;//compares String[] with List
12      ...
13      if_icmplt    L13;
14      getstatic    Field .../System.out:"Ljava/io/PrintStream;";
15      ldc  String "Success";
16      invokevirtual    Method ...println:"(Ljava/lang/String;)V";
17      return;
18 }
```

Figure 8: OpenJ9 Bug#12819

the effectiveness of the MCMC-based testing process in *classming*. That further motivates a promising direction of improving JavaTailor through designing more effective strategies to guide the process of test program synthesis, which will be discussed in Section 5.

4.3.4 RQ4: Unknown Bugs detected by JavaTailor. We also applied JavaTailor to test the latest build of both HotSpot and OpenJ9 for two most popular OpenJDK versions (i.e., OpenJDK8 and OpenJDK11). During the test time of five days, JavaTailor detects 10 unknown bugs and 6 have been confirmed or fixed by developers, while classming and Javaming did not detect any unknown bugs. One possible reason could be that classming has applied against the JVMs before, making them immune to the classming-like approaches. Table 4 shows the detailed information for the confirmed/fixed unknown bugs detected by JavaTailor. In particular, all these bugs cannot be detected by *classming* and *Javaming* based on our used benchmarks during the given testing period. We then used one unknown bug as en example for further illustration.

Figure 8 shows an OpenJ9 bug (Bug#12819 [3]) detected by JavaTailor, which is represented by Jasm (a bytecode-like assembly language that allows testers to reorganize bytecode order in a specific way [1]). In this example, Lines 14-16 corresponding to the `System.out.println("Success")` statement in the seed program, which should be executed and output `"Success"`. However, this statement was not executed on the latest OpenJ9 (for both OpenJDK8 and OpenJDK11) due to a bug in the OpenJ9's optimizer. Lines 9-13 of the inserted code contain a complex nested loop. Due to the space limit, we only showed the outermost loop. OpenJ9 optimizes this complex loop for better execution performance. Specifically, JVM stores the first parameter (if exists) of the `static` function (i.e., `String[]` in the `main` function at Line 1) to the local variable table at index 0, and then this local variable is overwritten by an `ArrayList` at Line 7. Since the optimizer of OpenJ9 assumes that the types of these parameters in the local variable table will not change during execution, it compares `String[]` with `List` at Line 11 but actually should compare `ArrayList` with `List`. Then, the optimizer believes that the `checkcast` must fail due to impossible conversion between `String[]` and `List`, and thus removes all the instructions after the outermost loop, causing that `"Success"` cannot be outputted. The developers of OpenJ9 fixed this bug by making the function's parameters in the local variable table become changeable during the optimization process, since these types may be changed during execution. Note that *classming* failed to detect this bug, since 1) *classming* cannot introduce such a complicated loop into the seed program; 2) it cannot introduce the instructions that overwrite the local variable table into the seed program.

5 DISCUSSION

5.1 Future Work

First, as presented in Section 4.3.3, regarding *classming*, MCMC-based mutation is more effective than random mutation, and thus it may be helpful to improve the performance of JavaTailor by designing an effective strategy for guiding synthesis. The strategy should guide to select both an ingredient and a seed program, as well as the synthesis point in the seed program. Second, in our study we came across one common but interesting type of false positives, i.e., HotSpot and OpenJ9 have different implementations for the same OpenJDK specification, but both of them believe they conform the specification. The root cause may lie in that the specification is a bit general. In the future, we may report such kind of inconsistencies to OpenJDK for further understanding. Third, although we evaluated JavaTailor on JVM implementations, the idea of JavaTailor is actually general. This idea could be generalized to other software taking programs as test inputs, such as compilers, symbolic executors, and database, as long as there are a large number of test programs revealing historical bugs that can be collected.

5.2 Threats to Validity

The internal threat to validity mainly lies in the implementations of JavaTailor and *classming*. To reduce this kind of threat, two authors carefully checked all code and we implemented them based on the mature tool Soot. Regarding the re-implementation of *classming*,

we implemented it according to the description in its paper and checked its correctness by reproducing its original evaluation.

The external threat to validity mainly lies in the benchmarks and historically bug-revealing test programs used in our study. In our study, we used six benchmarks from the existing study [30] and constructed two benchmarks based on the test suites of HotSpot and OpenJ9. Also, we collected 630 test programs revealing historical HotSpot's bugs from its test suite. To further reduce this kind of threat, we will evaluate JavaTailor on more benchmarks and the test programs revealing other JVM's bugs.

The construct threat to validity mainly in the randomness involved in those approaches. To reduce this threat, we conducted five differential-testing experiments instead of repeating one experiment several times. Indeed, our results demonstrate that JavaTailor stably outperforms *classming* in all the five experiments.

6 RELATED WORK

Since the first work on fuzzing [40], various techniques have been proposed for fuzzing software systems from different application domains [13, 32, 49, 51, 52, 55]. While all such techniques are related to this work, we mainly talk about the most closely related work in the areas of JVM testing and compiler fuzzing in this section.

JVM Testing. Due to the crucial role of JVM, both industry and academia proposed various testing techniques to ensure JVM's quality [11]. Besides *classming* and *classfuzz* introduced before, Sirer et al. [44] proposed *lava*, which generates test programs based on the production grammar. Yoshikawa et al. [54] proposed a random test program generator to test the JIT compiler. Freund et al. [33] developed a specification to verify bytecode verifiers in the form of a type system. Calvagna et al. [15–17] used a finite state machine model of the JVM specification to assess the conformance of JVM. Hwang et al. [36] proposed *JUSTGen*, which designs a set of domain specific languages and generates test programs by identifying unspecified cases from the JNI specification.

All of them except *classming* and *classfuzz* target one component in JVM. Different from them, JavaTailor is independent of a certain component in JVM. For example, our collected historical bug-revealing test programs for ingredient extraction cover the testing of a wide range of JVM components, such as C1 (client compiler), C2 (server compiler) and GC (garbage collection). Different from *classming* and *classfuzz*, JavaTailor synthesizes the ingredients extracted from historical bug-revealing test programs with a seed program to produce valid new test programs, and our experimental results have shown that JavaTailor significantly outperforms the state-of-the-art *classming*. In actual, JavaTailor can be combined with existing research, such as concurrency testing [48, 50], to target different components in JVM.

Compiler Testing. Similar to JVM, the test inputs of compilers are also programs [14, 21, 24], and thus we also briefly introduce the related work on compiler testing [19, 20, 22, 25, 27, 28, 43]. For example, Yang et al. [53] proposed *Csmith*, which generates C programs based on the grammar of the C language. Lidbury et al. [39] proposed *CLsmith* on the basis of Csmith for OpenCL compiler test program generation. Chen et al. [26]. proposed *HiCOND*, which uses historical data to infer a set of bug-revealing test configurations for effective test program generation. Le et al. [37] proposed *EMI*, which generates a program variant equivalent to the original test program under the given test inputs, and then uses these program pairs to test compilers. Based on the idea of EMI, researchers further proposed Athena [38] and Hermes [46]. Different from them, JavaTailor targets JVM testing by mining the ingredients in historical bug-revealing test programs and then inserting them to a seed program for test program generation. Such history-driven test program synthesis is also novel in the area of compiler testing and could be generalized to this area as discussed in Section 5.1.

7 CONCLUSION

In this paper, we propose a history-driven test program synthesis approach, called JavaTailor. It first tackles the challenge of extracting ingredients from bug-revealing test programs by designing five types of ingredients. Then, it inserts these extracted ingredients into seed programs, and automatically fixes the broken syntactic and semantic constraints in the ingredients in order to produce valid synthesized programs. Finally, these synthesized programs are used to differentially test JVMs. We conducted extensive experiments on popular JVM implementations (i.e., HotSpot and OpenJ9) to evaluate the effectiveness of JavaTailor. The experimental results demonstrate that JavaTailor significantly outperforms the state-of-the-art technique. That is, JavaTailor can achieve higher JVM code coverage and expose more unique inconsistencies. Moreover, JavaTailor found 6 unknown bugs that have been confirmed or fixed by developers.

ACKNOWLEDGMENT

We thank all the ICSE anonymous reviewers for their valuable comments. We also thank all the JVM developers for analyzing and replying to the bugs we reported. This work is partially funded by the National Natural Science Foundation of China Grant No. 62002256, 61872263, and the Tianjin Intelligent Manufacturing Special Fund Project Grant No. 20201180. This work is also partially supported by National Science Foundation under Grant Nos. CCF-2131943 and CCF-2141474, as well as Ant Group.

REFERENCES

[1] 2021. ASMTools. https://wiki.openjdk.java.net/display/CodeTools/asmtools.
[2] 2021. Bug-12552. https://github.com/eclipse-openj9/openj9/issues/12552.
[3] 2021. Bug-12819. https://github.com/eclipse-openj9/openj9/issues/12819.
[4] 2021. Gcov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[5] 2021. GIJ. https://web.archive.org/web/20070509055923/http://gcc.gnu.org/java.
[6] 2021. Hotspot. http://openjdk.java.net.
[7] 2021. JavaTailor. https://github.com/JavaTailor/CFSynthesis
[8] 2021. Lcov. http://ltp.sourceforge.net/coverage/lcov.php.
[9] 2021. OpenJ9. https://www.eclipse.org/openj9.
[10] 2021. Zulu. https://www.azulsystems.com/products/core.
[11] Bowen Alpern, Ton Ngo, Jong-Deok Choi, and Manu Sridharan. 2000. DejaVu: deterministic Java replay debugger for Jalapeño Java virtual machine. In *Object Oriented Programming Systems Languages and Applications Conference*. 165–166.
[12] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. 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*. 169–190.
[13] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2020. Fuzzing: Challenges and Reflections. *IEEE Software* (2020).
[14] Abdulazeez S. Boujarwah and Kassem Saleh. 1997. Compiler test case generation methods: a survey and assessment. *Inf. Softw. Technol.* 39, 9 (1997), 617–625.

[15] Andrea Calvagna, Andrea Fornaia, and Emiliano Tramontana. 2014. Combinatorial Interaction Testing of a Java Card Static Verifier. In *Seventh IEEE International Conference on Software Testing, Verification and Validation*. 84–87.

[16] Andrea Calvagna and Emiliano Tramontana. 2013. Automated Conformance Testing of Java Virtual Machines. In *Seventh International Conference on Complex, Intelligent, and Software Intensive Systems*. 547–552.

[17] Andrea Calvagna and Emiliano Tramontana. 2013. Combinatorial Validation Testing of Java Card Byte Code Verifiers. In *2013 Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*. 347–352.

[18] Felipe Canales, Geoffrey Hecht, and Alexandre Bergel. 2021. Optimization of Java Virtual Machine Flags using Feature Model and Genetic Algorithm. In *ICPE '21: ACM/SPEC International Conference on Performance Engineering*. 183–186.

[19] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to prioritize test programs for compiler testing. In *Proceedings of the 39th International Conference on Software Engineering*. 700–711.

[20] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. Test Case Prioritization for Compilers: A Text-Vector Based Approach. In *2016 IEEE International Conference on Software Testing, Verification and Validation*. 266–277.

[21] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler bug isolation via effective witness test program generation. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 223–234.

[22] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering*. 180–190.

[23] Junjie Chen, Haoyang Ma, and Lingming Zhang. 2020. Enhanced Compiler Bug Isolation via Memoized Search. In *35th IEEE/ACM International Conference on Automated Software Engineering*. 78–89.

[24] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1 (2020), 4:1–4:36.

[25] Junjie Chen and Chenyao Suo. 2022. Boosting Compiler Testing via Compiler Optimization Exploration. In *TOSEM*. to appear.

[26] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. 2019. History-Guided Configuration Diversification for Compiler Test-Program Generation. In *34th IEEE/ACM International Conference on Automated Software Engineering*. 305–316.

[27] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2021. Coverage Prediction for Accelerating Compiler Testing. *IEEE Trans. Software Eng.* 47, 2 (2021), 261–278.

[28] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. 2021. Efficient Compiler Autotuning via Bayesian Optimization. In *43rd IEEE/ACM International Conference on Software Engineering*. 1198–1209.

[29] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. 2020. Taming Behavioral Backward Incompatibilities via Cross-Project Testing and Analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 112–124.

[30] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *Proceedings of the 41st International Conference on Software Engineering*. 1257–1268.

[31] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 85–99.

[32] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. 2021. NTFUZZ: Enabling Type-Aware Kernel Fuzzing on Windows with Static Binary Analysis. (2021).

[33] Stephen N. Freund and John C. Mitchell. 2003. A Type System for the Java Bytecode Language and Verifier. *J. Autom. Reason.* 30, 3-4 (2003), 271–321.

[34] Vincenzo Gervasi and Roozbeh Farahbod. 2009. JASMine: Accessing Java Code from CoreASM. In *Rigorous Methods for Software Construction and Analysis, Essays Dedicated to Egon Börger on the Occasion of His 60th Birthday*, Vol. 5115. 170–186.

[35] Matthias Grimmer, Manuel Rigger, Roland Schatz, Lukas Stadler, and Hanspeter Mössenböck. 2014. TruffleC: dynamic execution of C on a Java virtual machine. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools*. 17–26.

[36] Sungjae Hwang, Sungho Lee, Jihoon Kim, and Sukyoung Ryu. 2021. JUSTGen: Effective Test Generation for Unspecified JNI Behaviors on JVMs. In *43rd IEEE/ACM International Conference on Software Engineering*. 1708–1718.

[37] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 216–226.

[38] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 386–399.

[39] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 65–76.

[40] Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44.

[41] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. 2009. Effective static deadlock detection. In *31st International Conference on Software Engineering*. 386–396.

[42] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing low-level languages to the JVM: efficient execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages*. 6–15.

[43] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). 968–980.

[44] Emin Gün Sirer and Brian N. Bershad. 1999. Using production grammars in software testing. In *Proceedings of the Second Conference on Domain-Specific Languages*. 1–13.

[45] James E. Smith and Ravi Nair. 2005. *Virtual machines - versatile platforms for systems and processes*. Elsevier.

[46] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 849–863.

[47] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research*. 13.

[48] Haichi Wang, Zan Wang, Jun Sun, Shuang Liu, Ayesha Sadiq, and Yuan-Fang Li. 2020. Towards Generating Thread-Safe Classes Automatically. In *35th IEEE/ACM International Conference on Automated Software Engineering*. 943–955.

[49] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). 788–799.

[50] Zan Wang, Yingquan Zhao, Shuang Liu, Jun Sun, Xiang Chen, and Huarui Lin. 2019. MAP-Coverage: A Novel Coverage Criterion for Testing Thread-Safe Classes. In *34th IEEE/ACM International Conference on Automated Software Engineering*. 722–734.

[51] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*.

[52] Ming Yan, Junjie Chen, Xiangyu Zhang, Lin Tan, Gan Wang, and Zan Wang. 2021. Exposing numerical bugs in deep learning via gradient back-propagation. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 627–638.

[53] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 283–294.

[54] Takahide Yoshikawa, Kouya Shimura, and Toshihiro Ozawa. 2003. Random Program Generator for Java JIT Compiler Test System. In *3rd International Conference on Quality Software*. 20.

[55] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 132–142.