Vectorizing Program Ingredients for Better JVM Testing

Tianchang Gao
College of Intelligence and Computing, Tianjin University
China
gaotc090@tju.edu.cn

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

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

JVM testing is one of the most widely-used methodologies for guaranteeing the quality of JVMs. Among various JVM testing techniques, synthesis-based JVM testing, which constructs a test program by synthesizing various code snippets (also called program ingredients), has been demonstrated state-of-the-art. The existing synthesis-based JVM testing work puts more efforts in ensuring the validity of synthesized test programs, but ignores the influence of huge ingredient space, which largely limits the ingredient exploration efficiency as well as JVM testing performance. In this work, we propose Vectorized JVM Testing (called VECT) to further promote the performance of synthesis-based JVM testing. Its key insight is to reduce the huge ingredient space by clustering semantically similar ingredients via vectorizing ingredients using state-of-the-art code representation. To make VECT complete and more effective, based on vectorized ingredients, VECT further designs a feedback-driven ingredient selection strategy and an enhanced test oracle. We conducted an extensive study to evaluate VECT on three popular JVMs (i.e., HotSpot, OpenJ9, and Bisheng JDK) involving five OpenJDK versions. The results demonstrate VECT detects 115.03%~776.92% more unique inconsistencies than the state-of-the-art JVM testing technique during the same testing time. In particular, VECT detects 26 previously unknown bugs for them, 15 of which have already been confirmed/fixed by developers.

CCS CONCEPTS

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

KEYWORDS

Java Virtual Machine, Program Synthesis, JVM Testing, Test Oracle

*Junjie Chen is the corresponding author.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598075

ACM Reference Format:
Tianchang Gao, Junjie Chen, Yingquan Zhao, Yuqun Zhang, and Lingming Zhang. 2023. Vectorizing Program Ingredients for Better JVM Testing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598075

1 INTRODUCTION

Java Virtual Machine (JVM) is one of the important infrastructures for the Java platform, and various Java applications require to be processed by JVMs for execution [25, 26, 46]. Over the years, many JVMs have been designed and developed by various organizations and companies, such as HotSpot from Oracle [4], OpenJ9 from IBM [5], GIJ from GNU [3], and Bisheng JDK from Huawei [1]. Although they are usually tested and maintained well, JVMs still contain bugs due to their quite complicated functionalities and huge code size [18]. In particular, JVM bugs could cause unexpected behaviors for any Java applications (and the programs that are written in other programming languages but can be compiled to Java bytecode) processed by them. Therefore, guaranteeing the quality of JVMs is definitely critical.

In the literature, some JVM testing techniques have been proposed [17, 18, 58], among which synthesis-based JVM testing is state-of-the-art and has outperformed the widely-studied mutation-based JVM testing techniques as demonstrated by prior work [58]. Specifically, synthesis-based JVM testing constructs a test program by synthesizing various code snippets, i.e., putting various ingredients extracted from historical bug-revealing test programs into a new context provided by a given seed program. Although it opens a promising direction for JVM testing, it is still in the initial stage and suffers from a major limitation. Specifically, the current synthesis-based JVM testing work puts more effort in ensuring the validity of synthesized test programs, but ignores the influence of huge ingredient space on the testing performance. Faced with huge ingredient space, the current synthesis-based JVM testing just conducts random search by treating each ingredient equally and individually, which can limit the ingredient exploration efficiency as well as the testing performance to a large extent.

To promote the performance of synthesis-based JVM testing, in this work, we propose a novel technique, i.e., Vectorized JVM Testing (called **VECT**). Its key insight is to improve the exploration of the huge ingredient space by measuring the semantic similarity

among ingredients via program ingredient vectorization. Specifically, ingredients are small code snippets tailored from complete test programs, and thus many of them are very likely to have similar semantics, leading to similar bug-revealing capabilities. With this intuition, VECT largely reduces the ingredient space by clustering the ingredients with similar semantics to the same group. In this way, ingredient selection from the huge space can be transformed to group selection from the smaller space, thus improving the ingredient exploration efficiency and the testing performance. In particular, VECT incorporates state-of-the-art code representations (e.g., CodeBERT [23], InferCode [11], CodeT5 [49], and PLBART [8]) to encode the ingredient semantics as vectors for ingredient clustering.

Although the search space can be largely reduced via ingredient vectorization and clustering, randomly selecting an ingredient group for test program synthesis still limits the testing performance. This is because different groups of ingredients can have different capabilities of generating bug-revealing test programs. Treating them equally may lead to the frequent generation of useless test programs. Hence, to make our proposed vectorized JVM testing technique effective, VECT then designs a feedback-driven group selection strategy based on the testing result after selecting an ingredient group. It aims to learn which groups can facilitate the generation of more bug-revealing test programs with fewer times of selection. Finally, differential testing is always employed by synthesis-based JVM testing as the test oracle, which compares the outputs of the test program across several JVMs. However, it can hardly ensure that the dependency between the ingredients and the output variables in the seed program is established after synthesis. Hence, it could lead to missing the detection of the bugs that are triggered by the synthesized test program but are not captured by the test oracle (through the output variables). To make the synthesized test programs really take effect for JVM testing, VECT monitors the results of various intermediate variables in the synthesized test program, and computes the checksum as the program output used by differential testing.

To evaluate the effectiveness of VECT, we conducted an extensive study on three popular JVM implementations (i.e., HotSpot [4], OpenJ9 [5], and Bisheng JDK [1]) involving five OpenJDK versions. The experimental results show that VECT can detect more unique inconsistencies than the state-of-the-art synthesis-based JVM testing technique (i.e., JavaTailor [58]) during the same testing time, achieving 115.03%~776.92% improvements across all the OpenJDK versions. In particular, VECT detects 26 previously unknown bugs in the latest HotSpot, OpenJ9, and Bisheng JDK, among which 15 bugs have already been confirmed or fixed by developers. The results demonstrate the effectiveness of VECT by vectorizing ingredients for speeding up the ingredient exploration.

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

- We propose Vectorized JVM Testing (i.e., VECT), which not only reduces the huge ingredient space by semantically vectorizing ingredients via code representation but also enhances the ingredient selection process and test oracle.
- We conducted an extensive study on three popular JVM implementations, demonstrating the effectiveness of VECT. In particular, VECT detected 26 previously unknown bugs, 15 of which have been confirmed or fixed by developers.

```
1  public static void main(String[] var0) {
2      test();
3      System.out.println(CHECKSUM);
4  }
5  public static void test() {
6      for(int var2 = 0; var2 < 100; ++var2) {
7          for(int var3 = var2; var3 < 100; ++var3) {
8              long var4 = 0;
9              int var6;
10             do {
11                 ++var4;
12                 long var5;
13                 var6 = ( var5 = var4 - 100) == 0 ?
14                             0 : (var5 < 0 ? -1 : 1);
15                 CHECKSUM = Check.checksum(CHECKSUM , var4);
16             } while(var6 < 0);
17         }
18     }
19 }
```

Figure 1: HotSpot bug JDK-8290451

```
int i = 1;
args = new int[2];
if(i < args.length){
    args[i] = args[i] >> 15;
    i++;
}
```

Ingredient-1

```
int i = 1;
args = new int[3];
if(i < args.length){
    args[i] = args[i] << 15;
    i++;
}
```

Ingredient-2

```
if(var1 >= 997){
    test_2vi(...);
    var2 = 0;
} else {
    var2 = var2 + verify(...);
    var1++;
}
```

Ingredient-3

```
if(var1 >= 3){
    var2 = 0;
} else {
    var2 = var2 + verify(...);
    var1++;
}
```

Ingredient-4

Figure 2: Ingredient examples

- We developed a tool to implement VECT, and released it as well as our experimental data at our project homepage [7] for replication, future research, and practical use.

2 MOTIVATING EXAMPLE

In this section, we use an example to motivate our work. Figure 1 shows a HotSpot bug (ID: JDK-8290451, which produces incorrect results when switching to C2 on-stack replacement compilation from C1) detected by our proposed technique (to be introduced in Section 3). The bug-revealing test program is synthesized by merging the ingredient (the code with blue shadow in Figure 1) into the seed program (the code without shadow in Figure 1). Based on the same seed program and ingredient pool, the state-of-the-art synthesis-based JVM testing technique, i.e., JavaTailor [58], could also generate the bug-revealing test program (without Lines 3 and 15 that are used for checksum calculation) if the proper ingredient is selected. The main contribution of JavaTailor is to construct *valid* test programs by fixing the broken syntactic and semantic constraints due to ingredient insertion, but it conducts random search for the ingredient space. Due to the huge ingredient space, selecting the proper ingredient in a random way is definitely inefficient. Moreover, the testing resource is limited, and thus poor efficiency could lead to poor effectiveness (e.g., missing the detection of this bug within a given testing time).

For ease of understanding, we simplifies the testing process by assuming that there are five ingredients in the pool in total. Figure 2

shows the other four ingredients (which cannot help construct the bug-revealing test program), and we call the bug-reveling ingredient (shown in Figure 1) *Ingredient-5*. If we enumerate each ingredient for test program synthesis, the selection times is five at the worst case. However, JavaTailor randomly selects an ingredient with replacement, the selection times could be more than five. Through observation, we can find that Ingredient-1 and Ingredient-2 are similar while Ingredient-3 and Ingredient-4 are similar in semantics. These ingredients with similar semantics are more likely to have similar bug-revealing capabilities. Indeed, similar ingredients are common in practice since ingredients are not complete programs but small code snippets tailored from complete test programs. If we can put the semantically similar ingredients into the same group, the ingredient space can be largely reduced as the group space. In this way, the testing efficiency can be improved. With this intuition, the five ingredients can be divided into three groups: {Ingredient-1, Ingredient-2}, {Ingredient-3, Ingredient-4}, and {Ingredient-5}. Then, through enumeration, the selection times is just three from the group space at the worst case. **This motivates that clustering semantically similar ingredients is an effective method of reducing the ingredient space, thus improving the testing efficiency.**

Although the ingredient space can be reduced after clustering, randomly selecting a group by treating them equally for test program synthesis still limits the testing efficiency. For example, assuming that the random strategy selects the first group at the first iteration, it cannot construct the bug-revealing test program. Intuitively, we prefer not to select this group at the next iteration. However, the random strategy also has the same probability to select this group at the next iteration. If we can utilize the selection history to guide the subsequent selections (e.g., we assign a smaller selection probability to this group at the next iteration according to the testing result of the first selection), it could be more efficient to select the bug-revealing one (i.e., the third group). Moreover, if we find that some groups contain bug-revealing ingredients according to the testing history, we should assign a larger selection probability to the group at the next iteration, which may help enhance the overall bug detection effectiveness. **This motivates that taking the selection and testing history as feedback can effectively guide future group selections, thus improving the overall testing efficiency.**

Actually, even though JavaTailor is able to synthesize the bug-revealing test program by giving enough testing time, it still cannot detect this bug. This is because it determines whether a test program detects a bug by comparing the results of the output variables on several JVMs (including checking crashes). However, the original seed program does not have any output variables, and thus this bug cannot be captured via output comparison. Such a case is common in practice, and meanwhile it is possible that there is no dependency between the variables in the ingredient and the output variables in the seed program, which may also cause that the triggered bug is not captured via output comparison. In this example, when monitoring the result of `var4` at Line 15 (or its affected variables, i.e., `var5` and `var6`), this bug can be captured via differential testing. **This motivates that due to the characteristics of test program synthesis, it is necessary to monitor the results of various intermediate variables for differential testing, thus enhancing the bug capture capability of the test oracle.**

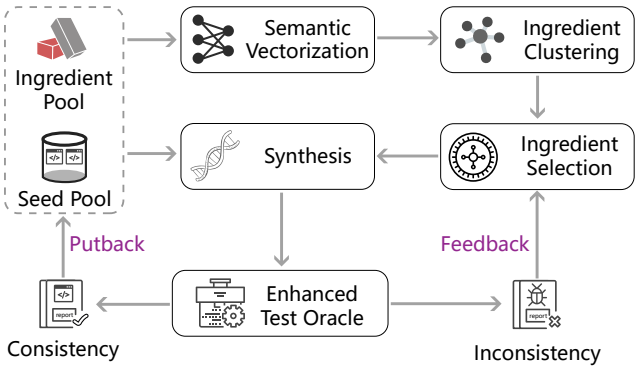

Figure 3: Overview of VECT

3 APPROACH

In this work, we propose a novel Vectorized JVM Testing technique, **VECT**, to promote the direction of synthesis-based JVM testing. Figure 3 shows the overview of VECT. It first vectorizes the semantics of ingredients through code representation (Section 3.1), and then clusters the ingredients with similar semantics into the same group (Section 3.2), which can largely reduce the search space for ingredient selection. Next, VECT gradually learns which groups of ingredients should be selected with higher probabilities based on the group selection history and the testing results of generated test programs during the testing process (Section 3.3). It can help guide the subsequent ingredient selections in order to achieve better testing performance. Finally, VECT designs an enhanced test oracle through monitoring the results of various intermediate variables in order to capture the JVM bugs triggered by the synthesized test programs more sufficiently (Section 3.4).

3.1 Ingredient Semantic Vectorization

As demonstrated by the existing work [58], five categories of ingredients have been considered by synthesis-based JVM testing and they are indeed effective. They are Sequential Ingredients, If Ingredients, Loop Ingredients, Switch Ingredients, and Try-Catch Ingredients. A sequential ingredient refers to a sequence of instructions without any branches in the test program, and the other categories of ingredients correspond to the program structure of `if-else`, loop (i.e., `while`, `do-while`, or `for`), `switch`, and `try-catch`, respectively. Following them, VECT also considers the five categories of ingredients for JVM test program synthesis. Due to the rich ingredient categories and a number of historical bug-revealing test programs as the ingredient providers, a large number of ingredients can be collected to form the huge ingredient space for JVM test program synthesis. The huge ingredient space can limit the testing performance, since it is challenging to select proper ingredients from the huge space to synthesize bug-revealing test programs.

Although ingredient providers often have different semantics, the ingredients (which are small code snippets tailored from complete test programs) may share similar semantics with each other,

thus leading to similar bug-revealing capabilities. With this intuition, VECT reduces the ingredient space by clustering the ingredients with similar semantics to the same group. That is, the huge ingredient space can be reduced as the smaller group space. Therefore, one key task in VECT is to precisely encode the semantics of each ingredient.

Instead of manually designing code features to vectorize code semantics, various code representation models have been proposed and constructed in recent years [8, 11, 23, 49]. The existing studies have demonstrated their effectiveness in many software engineering tasks, such as code search [11, 23, 27] and code clone detection [8, 11, 49]. Following the state-of-the-art practice, VECT also adopts code representation to encode ingredient semantics as vectors. In practice, there are a large number of advanced pre-trained code representation models. For the sake of cost-effectiveness, VECT directly borrows the power from the community of code representation by utilizing these pre-trained models to solve our task (i.e., ingredient semantic vectorization), similar to the existing work [48]. That is, our work does not pay attention to designing a new code representation method, but looks for a proper pre-trained code representation model to help achieve our goal.

Specifically, we investigated four state-of-the-art pre-trained code representation models in VECT for ingredient semantic vectorization, and conducted an experiment to explore their influence on the performance of VECT, which will be presented in Section 4.5.1. They are CodeBERT [23], InferCode [11], CodeT5 [49], and PLBART [8]. In particular, we carefully considered their diversity during selection, e.g., (1) InferCode transforms code as AST for representation, while the other three take code as token stream for representation; (2) they are based on different neural network structures. In detail, CodeBERT is based on the RoBERTa structure [37] and employs Masked Language Modeling [20] and Replaced Token Detection [19] as pre-training tasks. InferCode is an AST-based self-supervised pre-training model based on the TBCNN structure [38] by taking AST subtree prediction as the pre-training task. CodeT5 [49] is based on the T5 structure [41] and designs a novel identifier-aware pre-training task. PLBART [8] is based on the BART structure [34] and takes Token Masking, Token Deletion, and Token Infilling as pre-training tasks. More details about these models can be found in the corresponding papers. Indeed, it is hard to guarantee that there exists the optimal one for our task among the four models, and meanwhile VECT is not specific to the four models. Hence, in the future, we can investigate more code representation models in VECT to further improve its performance.

Regardless of code representation models, it can encode the semantics of an ingredient as a vector (called semantic vector in our paper). Since ingredients can be extracted at the Java-code or Jimple-code level (e.g., JavaTailor extracts ingredients from Jimple code), VECT should convert them to the Java-code level before code representation. This is because these code representation models are not trained with Jimple code data (but with Java code data) and meanwhile Java code and Jimple code differ significantly. Therefore, we can forecast that these pre-trained models cannot represent the semantics of Jimple code precisely.

3.2 Ingredient Clustering

Based on these semantic vectors of ingredients, VECT then clusters the ingredients with similar semantics into the same group. That is, the ingredients belonging to the same group are more likely to have the same (or similar) bug-revealing capabilities for synthesizing test programs. In VECT, we adopted the hierarchical clustering algorithm (i.e., AGNES [32], *AG*glomerative *NES*ting) to achieve the task of ingredient clustering. In fact, there are many clustering algorithms, but we choose AGNES due to the following three reasons: (1) The distribution of ingredients is extensive, and thus it is almost impossible to know the number of clusters in advance. Hence, the clustering algorithms that require to pre-define the number of clusters are not applicable, such as the widely-used K-means algorithm [28]. (2) The number of ingredients is large, and thus the clustering algorithms that perform not well on the large number of samples are not applicable, such as the density-based algorithms (e.g., DBSCAN [22]). (3) AGNES has a smaller number of parameters to be set than many widely-used clustering algorithms (e.g., K-means and DBSCAN), which makes it more easy-to-use in practice.

Given a set of ingredients denoted as $\mathcal{I} = \{i_1, i_2, ..., i_n\}$ (n refers to the number of ingredients) and the corresponding semantic vectors denoted as $\mathcal{V} = \{v_1, v_2, ..., v_n\}$, VECT adopts AGNES to produce a set of groups denoted as $\mathcal{G} = \{g_1, g_2, ..., g_r\}$, where g_i consists of n_i ingredients with similar semantics and r refers to the total number of groups. Specifically, AGNES utilizes the bottom-up aggregation strategy for clustering, which is an iterative process. In the initial stage, it regards each ingredient as a group, i.e., r is equal to n. Then, in each iteration, it calculates the distance between each pair of groups and merges the two closest groups into one group, until all the ingredients are put into the same cluster (i.e., r is equal to 1). The clustering process can be regarded as the process of building a hierarchical nested clustering tree from leaf nodes (the initial set of groups) to the root node (the group containing all the ingredients). Finally, it outputs the set of groups at a hierarchy in the tree with the highest quality as the final clustering result.

During the iterative clustering process, there are two core steps: (1) measuring the distance between two groups, and (2) measuring the quality of a set of groups at a hierarchy in the tree. Regarding the former, VECT calculates the average distance between each pair of ingredients from the two groups as shown in Formula 1.

$$d_{avg}(g_i, g_j) = \frac{\sum_{v_x \in g_i} \sum_{v_y \in g_j} dist(v_x, v_y)}{|g_i||g_j|} \tag{1}$$

where $dist(v_i, v_j)$ calculates Euclidean distance between the semantic vectors corresponding to a pair of ingredients from g_i and g_j. Regard the latter, VECT adopts Silhouette Coefficient [43] as the metric following the existing work [44]. It measures how similar a sample is to its own group (cohesion) compared to other groups (separation). The closer its value is to 1, the higher the clustering quality is. Its definition is shown in Formula 2.

$$SC = \frac{1}{n} \sum_{k=1}^{n} \frac{b_k - a_k}{max\{a_k, b_k\}} \tag{2}$$

Algorithm 1: Ingredient Selection

```
Input: 𝒢: As set of groups obtained from ingredient clustering
Output: I: The selected ingredient
1  Function IngredientSelection(𝒢):
2      𝒲 ← [] ; /* The list of weights for groups */
3      for g_i in 𝒢 do
4          𝒲.add(weight(g_i)) ;
5      𝒫 ← [] ; /* The list of probabilities for groups*/
6      for w_i in 𝒲 do
7          𝒫.add(w_i/∑_{j=1}^{r} w_j);
8      g ← SelectGroup(𝒢, 𝒫);
9      I ← RandomIngredient(g); /* randomly select an ingredient
         from g*/
10     return I;
11 Function SelectGroup(𝒢, 𝒫):
12     index ← 0 ;
13     r ← random(0, 1) ;
14     for p_i in 𝒫 do
15         if p_i > r then
16             return 𝒢.get(index) ;
17         else
18             r ← r - p_i ;
19             index ← index + 1 ;
```

where n refers to the total number of ingredients, a_k represents the average distance between the ingredient i_k and the other ingredients in the same group, and b_k represents the average distance between the ingredient i_k and the ingredients in the other groups. With this metric, VECT can obtain the clustering result with the highest quality (closest to 1).

3.3 Ingredient Selection

After clustering, VECT requires to select ingredients for synthesizing new test programs. Although the ingredient space is reduced as the group space, random group selection is still a less efficient strategy. To improve the testing performance, VECT designs a feedback-driven group selection strategy. VECT considers two criteria in the selection strategy: (1) *Bug-revealing capability.* The groups that can facilitate to synthesize bug-revealing test programs should be selected with larger probabilities. (2) *Diversity.* The groups that were selected more rarely should be selected with larger probabilities. Algorithm 1 formally illustrates the selection process of VECT.

Based on the two criteria, VECT records which group is selected for generating a new test program and whether the test program detects a JVM inconsistency, in order to guide subsequent group selections along with the testing process. Specifically, it calculates the weight that a group is selected for test program synthesis as shown in Formula 3 (Lines 2-4 in Algorithm 1).

$$weight(g_i) = \frac{t_i + 1}{s_i + 1} (1 \leq i \leq r) \tag{3}$$

where s_i represents the total number of times that the group g_i is selected during the testing process, and t_i represents the total number of times that the generated test program after selecting g_i triggers a JVM inconsistency. Here, the "1" on the numerator and denominator indicates that the initial selection weight of each group is 1.

Then, VECT normalizes the selection weight of each group based on the Roulette Wheel algorithm [47] to balance the scale of different weights (Lines 5-7). Formula 4 presents the normalization process.

$$p_i = \frac{weight(g_i)}{\sum_{j=1}^{r} weight(g_j)} \tag{4}$$

where r is the number of groups and p_i represents the selection probability of the group g_i.

According to the selection probability of each group, before constructing a test program, VECT first selects a group (Line 8). However, an ingredient (rather than a group) is required by test program synthesis, and thus VECT should further select an ingredient from the group. Since the ingredients in the same group have similar semantics, VECT randomly selects one ingredient from it as the one used for test program synthesis (Line 9). Here, we do not fix an ingredient as the representative of a group for test program synthesis, since repeatedly using one ingredient in the selected group could lead to less diversity than the random ingredient selection method.

After selecting an ingredient through the feedback-driven strategy, VECT constructs a new test program by synthesizing the ingredient into the given seed program. Due to the complex syntactic and semantic constraints involved in a program, the synthesis could break those constraints (e.g., incurring undefined variables), leading to an invalid test program. To generate a valid test program, VECT adopts the constraint fixing strategy proposed in JavaTailor.

3.4 Enhanced Test Oracle

The existing JVM testing techniques (including the state-of-the-art JavaTailor) employ differential testing as the test oracle [17, 18, 58]. Specifically, they compare the outputs of the same test program on several JVMs (e.g., HotSpot and OpenJ9). If the outputs are different, it indicates that the test program triggers an inconsistency among JVMs. They consider two kinds of outputs: (1) The exception messages or crash messages when the test program terminates abnormally; (2) The execution results (produced based on the output variables in the test program) when the test program terminates normally. However, when synthesizing an ingredient into a seed program, the ingredient is very likely to have no dependency with the output variables in the seed program. Therefore, the inconsistency triggered by the synthesized test program could be not captured by the current widely-used test oracle. That is, when an inconsistency is triggered and captured by the variables in the ingredient, it may be not propagated to the output variables due to without dependency between them.

To improve the performance of synthesis-based JVM testing, VECT enhances the test oracle by considering the characteristics of test program synthesis. That is, the enhanced test oracle, instead of just observing the results of the output variables in the seed program, monitors the results of various intermediate variables in the synthesized test program. In this way, the program states affected by the ingredient can be also reflected from the outputs for differential testing. Specifically, instead of instrumenting a number

of `print` statements for recording the results of intermediate variables (which can incur extra I/O overheads), VECT calculates the checksum of intermediate variables in the synthesized test program as the outputs inspired by the existing work [54].

In fact, adding extra checksum calculations could negatively affect the execution efficiency of synthesized test programs. Hence, VECT designs some compromises in the enhanced test oracle to balance the testing effectiveness and efficiency. First, VECT considers all the global and local variables with primitive types for checksum calculation, but ignores the variables with reference types. This is because the fields of a reference-type variable could be other reference types, which would cause recursive access until finding primitive types. When the reference-type dependency is long/circular, this process could be costly, even causing `StackOverFlowError`. Second, the value of each variable tends to be updated several times in a test program and thus performing checksum calculation for each update could be costly. Hence, VECT identifies the last access for each variable through static analysis and then performs checksum calculation by using the final value of each variable. In addition, some variables involve non-determinism, such as randomness and timestamps, which can incur inaccuracies to the enhanced test oracle and thus lead to false positives. Therefore, VECT discards them from checksum calculations by analyzing the dependency between variables and non-deterministic instructions (e.g., `System.currentTimeMillis` and `Random().nextInt`).

With the enhanced test oracle, when the synthesized test program terminates normally, VECT can better capture the triggered inconsistency through checksum outputs. Moreover, same as the existing practice [58], when the test program terminates abnormally, VECT also captures the inconsistency according to the exception messages or crash messages. Despite our best effort for reducing noise in differential testing (such as filtering out non-deterministic outputs), we still manually analyze whether a found inconsistency is a real bug or a false positive before reporting it to JVM's developers. If an inconsistency is a false positive, we then design a rule accordingly and incorporate it in VECT to improve the accuracy of subsequent JVM testing. If a synthesized test program does not detect any inconsistency and terminates normally, VECT puts it into the seed pool for high-order synthesis.

4 EVALUATION

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

- **RQ1**: Which code representation model is the most effective to VECT?
- **RQ2**: How does VECT perform in detecting JVM bugs?
- **RQ3**: Can VECT achieve higher JVM code coverage?
- **RQ4**: Does each main component in VECT contribute to its overall performance?

4.1 JVMs, Seed Programs, and Ingredients

JVMs. In the study, we adopted three popular JVMs as subjects, including the widely-studied ones in the existing work [17, 18, 58] (i.e., HotSpot [4] and OpenJ9 [5]) and an emerging JVM in recent years (i.e., Bisheng JDK [1]). For fair comparison with the state-of-the-art synthesis-based JVM testing technique (i.e., JavaTailor), we used *the same versions* for HotSpot and OpenJ9 as the evaluation of JavaTailor. As shown in Table 1, we considered five OpenJDK versions for them. Specifically, for each OpenJDK version, we first used the old builds for differential testing, since these old builds tend to contain more bugs that can help obtain more significant comparison results in statistics. Then, to investigate whether VECT can detect previously unknown JVM bugs, we also used OpenJDK-8 and OpenJDK-11 as the representative, and then applied VECT to the new builds of the two OpenJDK versions for differential testing, respectively. Since Bisheng JDK was not used by the existing study [58], we selected an old build and the latest build for OpenJDK-8 and OpenJDK-11, respectively. Please note that Bisheng JDK does not release builds for OpenJDK-12, OpenJDK-13, and OpenJDK-14.

Table 1: Studied JVM versions

OpenJDK Version	JVM Implementation	Version
OpenJDK8	HotSpot	bulid 25.0-b70
		build 25.345-b01
	OpenJ9	bulid openj9-0.8.0
		build openj9-0.32.0
	Bisheng JDK	build 25.302-b13
		build 25.332-b11
OpenJDK11	HotSpot	bulid 11+2
		build 11.0.16+8
	OpenJ9	build openj9-0.12.0
		build openj9-0.32.0
	Bisheng JDK	build 11.0.12+13
		build 11.0.15+11
OpenJDK12	HotSpot	build 12+33
	OpenJ9	build openj9-0.13.0
OpenJDK13	HotSpot	build 13+33
	OpenJ9	build openj9-0.16.0
OpenJDK14	HotSpot	build 14+36-1461
	OpenJ9	build openj9-0.20.0

Shadow represents the used new build of the corresponding JVM.

To sum up, we performed seven differential-testing experiments among HotSpot, OpenJ9, and Bisheng JDK in total: (1) five differential-testing experiments on old builds of these JVMs due to five OpenJDK versions; (2) two differential-testing experiments on new builds of these JVMs for OpenJDK-8 and OpenJDK-11. Table 1 shows the specific version/build information in our study.

Seed Programs. For fair comparison, we used the same benchmarks as the evaluation of JavaTailor [58]. Table 2 shows the basic information of these benchmarks. Each of the first six benchmarks contains only one seed program (i.e., the *classfile* including the `main` function). The latter two benchmarks contain more test programs, which are the historical bug-revealing test programs collected from

Table 2: Benchmarks

ID	Project	#Size	#Inst	#Time
P1	avrora	1	294	39,017 s
P2	eclipse	1	2,057	144,000 s
P3	pmd	1	806	86,400 s
P4	jython	1	369	120,783 s
P5	fop	1	186	17,168 s
P6	sunflow	1	305	34,809 s
P7	HotSpot-tests	563	116,427	3 days
P8	OpenJ9-tests	653	246,370	3 days

the repositories of HotSpot and OpenJ9, respectively. Since the enhanced test oracle of VECT calculates checksum of intermediate variables, the seed programs involving the multi-thread mechanism were removed from the benchmarks. Hence, the number of seed programs in the latter two benchmarks is smaller than that used in the evaluation of JavaTailor. In Table 2, Column *#inst* represents the total number of Jimple instructions of the seed programs in each benchmark and Column *#time* represents the testing time spent on each benchmark set by the existing work for evaluating JavaTailor [58]. For fair comparison, we also ran each studied JVM testing technique for the same testing time by taking the corresponding benchmark as the seed programs.

Ingredients. Same as the existing work [58], we also extracted five categories of ingredients from the P7 benchmark (a set of historical bug-revealing test programs for HotSpot) as the ingredient pool in our study. In total, there are 31,571 ingredients, including 16,973 Sequential ingredients, 8,362 If ingredients, 5,812 Loop ingredients, 397 Try-Catch ingredients, and 27 Switch ingredients

4.2 Compared Techniques

Since VECT also belongs to synthesis-based JVM testing, we compared it with the state-of-the-art synthesis-based technique, i.e., **JavaTailor**. JavaTailor randomly selects an ingredient from the ingredient pool to synthesize a new test program and then adopts differential testing based on the output variables in the seed program (also including exception or crash messages when the test program terminates abnormally) as the test oracle. Although there are some other JVM testing techniques, such as *classming* [17], the evaluation of JavaTailor has demonstrated the superiority of JavaTailor over them. Moreover, our evaluation shares the same experimental setting as that of JavaTailor. Hence, we did not consider the other techniques in our study.

To answer RQ1, we compared the performance of VECT with different code representation models. As presented in Section 3.1, we investigated four typical code representation models, i.e., CodeBERT, InferCode, CodeT5, and PLBART. That is, we constructed four instantiations of VECT. For ease of presentation, we call them $\textbf{VECT}_{\textit{CodeBERT}}$, $\textbf{VECT}_{\textit{InferCode}}$, $\textbf{VECT}_{\textit{CodeT5}}$, and $\textbf{VECT}_{\textit{PLBART}}$, respectively. Then, we used the most effective instantiation as the default VECT in the experiments of answering the other RQs and the practical use.

To answer RQ4, we constructed three variants of VECT to investigate the contribution of different components (i.e., our ingredient representation and clustering component, our feedback-driven ingredient selection component, and our checksum-based test oracle). For ease of presentation, we call them $\textbf{VECT}_{\textit{noCluster}}$ (removing our ingredient representation and clustering component), $\textbf{VECT}_{\textit{noFeedback}}$ (removing our feedback-driven ingredient selection component, but randomly selecting an ingredient group after clustering for synthesizing a new test program) and $\textbf{VECT}_{\textit{noChecksum}}$ (removing our checksum-based test oracle).

4.3 Measurements

Number of known unique inconsistencies. In the differential-testing experiments on old builds of JVMs, each studied technique may detect a number of inconsistencies during the same testing time. As presented before, an inconsistency may be a real bug or a false positive, and thus we further ran the test program triggering the inconsistency on the latest builds of JVMs in order to check whether the inconsistency still exists or not till now. If this inconsistency disappears, we regarded it as a known bug that has been fixed before the latest builds; Otherwise, we further manually investigated it to obtain the conclusion of the inconsistency and reported the potential bug to the developers of JVMs.

Through the above method, we can obtain a set of inconsistencies corresponding to known bugs, but these inconsistencies may be duplicate. Hence, we further de-duplicated them in two ways according to different inconsistency types. For the inconsistencies with exception or crash messages, we used the corresponding messages for de-duplication following the existing work [58]. For the inconsistencies detected when test programs terminate normally (e.g., those detected through the checksum outputs), we adapted the Correcting Commit method [13] for de-duplication. That is, it identifies the first build that fixes the bug (making the test program not produce the inconsistency again). If the same build is identified for two inconsistency-triggering test programs, we regarded that they detected the same known bug. Although this method may not be perfect (a threat in our study), the existing study has demonstrated its accuracy and meanwhile it is the only automatic method for de-duplication in the area of compiler testing. With the above methods, we can measure the number of unique inconsistencies corresponding to known bugs from a number of inconsistencies detected by a JVM testing technique.

Number of previously unknown bugs. In the differential-testing experiments on new builds of JVMs, each studied technique may also detect inconsistencies during the same testing time. Here, we used the same new builds as the evaluation of JavaTailor for fair comparison, which aims to avoid the influence that the JVM builds under test are immune to JavaTailor. That is, they may be not the latest build. Therefore, we further reproduced these inconsistencies on the latest build, in order to obtain a set of inconsistencies that still exist on the latest builds, which may be caused by previously unknown bugs. For these inconsistencies, the above-mentioned automatic method cannot be applied to determine whether they are real bugs or false positives. Hence, we manually analyzed them and then reported potential bugs to the developers of JVMs. Finally, we can measure the number of previously unknown bugs according to the developers' feedback.

Test coverage of JVM. We also measured the test coverage of JVM achieved by each technique during the same testing time in order to further understand the performance difference between

Table 3: Comparison results of VECT under different code representation models

Model	InferCode	CodeBERT	CodeT5	PLBART
#Unique Inconsistencies	28	15	26	45

them. Following the existing work [58], we measured line coverage, branch coverage, and function coverage, respectively.

4.4 Implementation and Environment

We implemented VECT in Java and Python. We adopted the pre-trained code representation models released by the corresponding work [8, 11, 23, 49] in VECT. We adapted the AGNES clustering algorithm to our task based on the sklearn library [6]. Regarding the code representation models and the clustering algorithm, we used their default parameters. For fair comparison, we ran each studied technique on each benchmark for the same testing time, and set the testing time budget on each benchmark following the existing work [58] as shown in the last column in Table 1.

In particular, we have developed a tool for VECT and *released it as well as our experimental data on our project homepage [7]*, for replication, future research, and practical use. All our experiments were conducted on a sever with Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz and 128GB RAM, running Ubuntu 16.04 LTS (64 bit).

4.5 Results and Analysis

4.5.1 RQ1: Influence of Different Code Representation Models. This experiment aims to select the best code representation model (among the four studied models) for VECT. Here, we used OpenJDK-11 and the P7 benchmark as the representative due to the large costs of running each instantiation of VECT on all the versions and benchmarks. Table 3 shows the comparison results among $\text{VECT}_{CodeBERT}$, $\text{VECT}_{InferCode}$, VECT_{CodeT5}, and VECT_{PLBART} in terms of the number of unique inconsistencies. We found that $\text{VECT}_{InferCode}$ (28) and VECT_{PLBART} (45) detect more unique inconsistencies than $\text{VECT}_{CodeBERT}$ (15) and VECT_{CodeT5} (26) during the same testing time. The main reason may be that the two pre-trained models (i.e., PLBART and InferCode) learn the Java code semantics better than the remaining two (i.e., CodeBERT and CodeT5). Specifically, the size of Java code training data for PLBART and InferCode is 470M and 16M respectively, while that for CodeT5 and CodeBERT is just 1.6M and 1.5M respectively. In general, the quantity and quality of training data can largely affect the performance of the trained models [57]. Moreover, the pre-training task of InferCode may be more suitable to our task than the other three, since it considers the code structure information with AST analysis (besides textual information). Therefore, InferCode can achieve similar performance with PLBART even though the size of Java code training data for the former is significantly smaller than that for the latter. To sum up, *we selected PLBART as the default code representation model in VECT due to its effectiveness.*

4.5.2 RQ2: Performance of VECT in JVM Bug Detection. We compared VECT with JavaTailor in terms of the number of unique inconsistencies (corresponding to known bugs) and the number of previously unknown bugs to answer RQ2.

Comparison in terms of unique inconsistencies. Table 4 shows the comparison results between VECT and JavaTailor in terms of unique inconsistencies from differential-testing experiments on old builds. First of all, all the detected inconsistencies by JavaTailor are due to exceptions or crashes (i.e., the test programs terminate abnormally), which is consistent with the conclusion from the existing work [58]. That is, the output variables in the seed programs are ineffective to capture the bugs triggered by the synthesized test programs. In contrast, VECT can detect both the inconsistencies due to exceptions or crashes and the inconsistencies due to outputs from normal executions. In particular, the number of unique inconsistencies for the latter on all the benchmarks is significant (ranging from 62 to 123 in the five differential-testing experiments). This results demonstrate the contribution of our enhanced test oracle through checksum calculation, which largely improves the effectiveness of synthesis-based JVM testing.

In terms of the number of unique inconsistencies due to exceptions/crashes, VECT also outperforms JavaTailor in all the five differential-testing experiments. During the same testing time, the average number across the five experiments of VECT is 80.8, while that of JavaTailor is 51.2. The improvement of VECT over JavaTailor ranges from 43.93% to 223.08% in the five experiments. The results demonstrate that VECT speeds up the detection of JVM bugs compared with JavaTailor through reducing the ingredient space with code representation and clustering and designing a feedback-driven ingredient selection strategy.

Overall, during the same testing time, VECT detects 115.03% ~ 776.92% more unique inconsistencies (including both exception inconsistencies and output inconsistencies) than JavaTailor in the five differential-testing experiments.

Comparison in terms of previously unknown bugs. We further compared VECT with JavaTailor in terms of the number of previously unknown bugs from the differential-testing experiments on new builds. As demonstrated by the existing work [58], JavaTailor detected 6 previously unknown bugs (confirmed or fixed by developers) at that time of evaluating JavaTailor. Since we used the same JVM builds under test as the evaluation of JavaTailor, our experimental results further confirmed the conclusion of JavaTailor (i.e., the same 6 bugs). During the same testing time, all the 6 bugs were also detected by VECT, and meanwhile VECT detected 26 additional new bugs, 15 of which have been confirmed or fixed by developers. Table 5 shows the information of the 15 confirmed/fixed previously unknown bugs. Among them, 7 bugs are captured due to the inconsistencies based on exceptions/crashes. In theory, if we ran JavaTailor for enough time, they can be also detected by it since we used the same seed programs and ingredient pool for both VECT and JavaTailor. That further demonstrates that VECT is able to largely improve the efficiency of synthesis-based JVM testing. The remaining 8 bugs are captured by VECT due to the inconsistencies of checksum results, and thus they cannot be detected by JavaTailor even if longer testing time is provided. That further demonstrates that VECT is also able to improve the effectiveness of synthesis-based JVM testing.

Besides the bug used in Section 2, we further used another previously unknown bug detected by VECT as an example for further illustration. Figure 4 shows a simplified synthesized test program

Table 4: Comparison results of JavaTailor and VECT in terms of the number of unique inconsistencies

ID	OpenJDK8			OpenJDK11			OpenJDK12			OpenJDK13			OpenJDK14		
	J.T.	VECT		J.T.	VECT		J.T.	VECT		J.T.	VECT		J.T.	VECT	
	E.D.	E.D.	C.D.	E.D.	E.D.	C.D.	E.D.	E.D.	C.D.	E.D.	E.D.	C.D.	E.D.	E.D.	C.D.
P1	11	11	0	0	0	0	1	3	0	0	0	0	0	0	0
P2	10	15	0	1	1	0	0	1	0	1	0	0	1	14	0
P3	7	13	1	0	2	1	2	2	1	0	2	0	0	0	0
P4	6	7	0	0	1	0	0	1	0	0	0	0	0	0	0
P5	3	3	0	0	0	0	0	2	0	0	0	0	0	0	0
P6	2	9	54	1	1	52	2	6	55	1	3	38	0	1	66
P7	42	55	37	12	14	31	10	14	5	2	8	16	3	5	52
P8	92	136	31	14	22	23	14	15	3	9	15	8	9	22	9
Total	**173**	**249**	**123**	**28**	**41**	**107**	**29**	**44**	**64**	**13**	**28**	**62**	**13**	**42**	**72**

J.T. : JavaTailor E.D. : Exception/Crash Difference C.D. : Checksum Difference

Table 5: Previously confirmed or fixed bugs detected by VECT

Bug ID	JVM	Affected OpenJDK	Status	Type
Bug#14716	OpenJ9	8,11,17,18	Fixed	Exception
Bug#15166	OpenJ9	8,11,17,18	Fixed	Exception
Bug#15500	OpenJ9	8,11,17,19	Confirmed	Checksum
Bug#16202	OpenJ9	11,17	Confirmed	Exception
JDK-8290451	HotSpot	8,11,17,19,20	Fixed	Checksum
JDK-8290705	HotSpot	8,11,17-20	Fixed	Checksum
JDK-8293044	HotSpot	8,11,17-20	Fixed	Checksum
JDK-8294889	HotSpot	11,17	Confirmed	Exception
JDK-8294938	HotSpot	8	Confirmed	Exception
Bug#I5IBSU	Bisheng JDK	8,11	Confirmed	Checksum
Bug#I5HV0D	Bisheng JDK	8,11	Confirmed	Checksum
Bug#I5HDPU	Bisheng JDK	8,11	Confirmed	Checksum
Bug#I5IBSU	Bisheng JDK	11	Confirmed	Checksum
Bug#I5XRFM	Bisheng JDK	8	Confirmed	Exception
Bug#I5XRFS	Bisheng JDK	11	Confirmed	Exception

```
1  public static int[] src = new int[1];
2  public static int[] dst = new int[536870913];
3  public static int CHECKSUM = 0;
4  public static void main(String[] var0) {
5      for(int var1 = 0; var1 < 20000; ++var1) {
6          test();
7      }
8      System.out.print(CHECKSUM);
9  }
10 public static void test() {
11     ...
12     int var0 = 536870912;
13     System.arraycopy(src, 0, dst, var0, 1);
14     System.arraycopy(dst, var0, src, 0, 1);
15     CHECKSUM = Check.checksum(CHECKSUM , var0);
16     ...
17 }
```

Figure 4: OpenJ9 bug #15500

triggering a JIT optimization bug in OpenJ9. This bug is detected via checksum comparison. In this example, VECT inserts an ingredient containing two invocations to the built-in array API (i.e., `System.arraycopy`) before Line 16, and inserts the checksum calculation of the intermediate variable introduced by the ingredient (i.e., `var0`) at Line 15. The ingredient and the checksum calculation statement (Lines 12-15) are identified as hot code and thus trigger JIT optimizations, since the function `test` they belong to is repeatedly executed 20,000 times at Line 6. However, OpenJ9 incorrectly assumes that the second `arraycopy` at Line 14 will trigger a bound check failure during local value propagation optimization, and thus removes all the statements after Line 14. The wrong optimization makes OpenJ9 produce incorrect checksum results compared to Hotspot. The developers of OpenJ9 have confirmed this bug and assured to fix it in the next release. Note that this bug cannot be detected by JavaTailor, since JavaTailor cannot observe the state of intermediate variables.

4.5.3 RQ3: Performance of VECT in JVM Code Coverage. To better understand why VECT detects more bugs than JavaTailor during the same testing time, we further compared them in JVM coverage achieved by them by taking OpenJDK-11 build 11+2 of HotSpot as the representative. Here, we collected JVM line coverage, branch coverage, and function coverage through the widely-used coverage collection tool (i.e., Gcov [2]). Due to the space limit, we just reported the comparison results in terms of line coverage as shown in Figure 5, and all the results can be found at our project homepage. Indeed, the conclusions from line coverage, branch coverage, and function coverage are consistent. In Figure 5, the x-axis represents each benchmark while the y-axis represents the line coverage achieved by each technique when taking the corresponding benchmark as the seed programs. From this figure, VECT achieves higher JVM line coverage than JavaTailor on each benchmark during the same testing time. That demonstrates the performance of VECT in terms of JVM coverage.

We further analyzed the coverage growth trend with the testing process proceeding, to better understand the efficiency superiority of VECT. Here, we used the P5 benchmark as the representative (which has the moderate improvement of VECT over JavaTailor in Figure 5). Figure 6 shows the trend, where the x-axis represents the testing time while the y-axis represents the achieved line

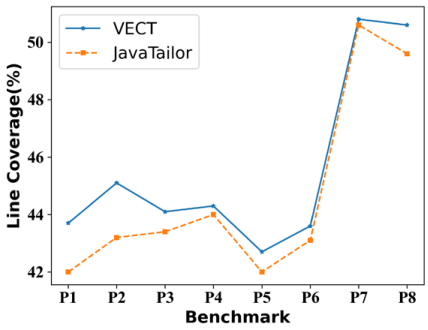

Figure 5: Line coverage

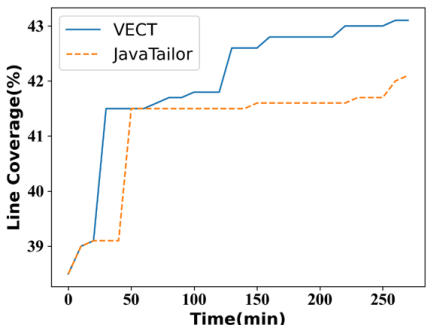

Figure 6: Line coverage trend on P4

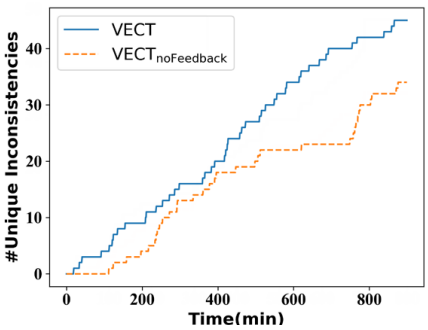

Figure 7: VECT v.s. VECT$_{noFeedback}$

coverage. From this figure, except at the very early stage, VECT always achieves higher JVM line coverage than JavaTailor, and the superiority of VECT becomes more obvious after one hour. This is because at the very early stage, the feedback-driven ingredient selection strategy does not learn too much for guiding the testing process. With the testing process proceeding, this strategy can receive more and more feedback and meanwhile the superiority of the reduced ingredient space becomes obvious gradually, and thus VECT achieves better results in terms of JVM line coverage accordingly. That demonstrates the high efficiency of VECT.

4.5.4 RQ4: Contribution of Each Main Component in VECT. We investigated the contribution of three main components in VECT by comparing VECT with VECT$_{noCluster}$, VECT$_{noFeedback}$, VECT$_{noChecksum}$, and taking OpenJDK11 and the P7 benchmark as the representative. Table 6 shows the comparison results among them in terms of the number of unique inconsistencies. From Table 6, we found that VECT performs better than all of the three variants, and they are all superior to JavaTailor. The results demonstrate the contribution of each component to the overall performance of VECT. The performance of VECT$_{noChecksum}$ is relatively poor since VECT without checksum-based test oracle can only find the unique inconsistencies due to exceptions/crashes.

Figure 7 further shows the comparison results between VECT and VECT$_{noFeedback}$, where the x-axis represents the testing time while the y-axis represents the number of unique inconsistencies detected by each technique within the corresponding testing time. From this figure, VECT detects more unique inconsistencies than VECT$_{noFeedback}$ during the entire testing process. With the testing process proceeding, the superiority of VECT over VECT$_{noFeedback}$ becomes more and more obvious. This is as expected, since there is no too much feedback that can be used by our feedback-driven selection strategy during the early stage, but the feedback information can be gradually accumulated. The results further demonstrate the contribution of our feedback-driven selection strategy in VECT.

5 DISCUSSION

5.1 Future Work

In the future, we can improve VECT from two aspects. First, VECT applies code representation to encode Java code semantics as vectors. Actually, Jimple code can be also treated as the test inputs of JVMs, and VECT converts Jimple code to Java code for semantic vectorization. The conversion process can incur extra costs. In the future, we may fine-tune the pre-trained code representation models with Jimple code so as to better fit this form of ingredients. Second, despite our best efforts for improving the bug detection accuracy of VECT, it still reports a few false positives. There are two main reasons: (1) Some contents in the OpenJDK specification are general, causing that different JVMs have different implementations for the same specification. However, they all believe that they conform the specification via communication. In the future, we will try to communicate with the designers of the specification, in order to further handle these cases. (2) The dependent packages (e.g., `java.lang.Math`) by the synthesized test programs may involve randomness, which can affect the accuracy of checksum comparison. Although VECT identifies non-deterministic instructions inside the test programs, it is non-trivial to identify such instructions inside all the dependent packages in advance. Therefore, after coming across this kind of issues, we designed and programmed some rules in VECT to avoid similar issues in future testing. In the future, we will incorporate advanced program analysis methods to more elegantly address this problem.

5.2 Threats to Validity

The *internal* threat to validity mainly lies in the implementation of VECT and JavaTailor. To reduce this threat, we directly adopted the implementation of JavaTailor released by the existing work [58]. VECT was implemented based on some mature libraries and pre-trained code representation models released by the corresponding work [8, 11, 23, 49]. Two authors have carefully check all our source code. We also released our implementation for replication. Note that, we designed more filtering rules (e.g., some implementations among JVMs are different but all correct, which may produce different exceptions for the same cases) and applied them to VECT and JavaTailor for obtaining more accurate results.

The *external* threats to validity mainly lie in the seed programs and ingredients used in our study. For fair comparison with JavaTailor, we adopted almost the same seed programs and ingredients as the existing study [58]. Here, we discarded the seed programs and ingredients involving the multi-thread mechanism to ensure the accuracy of the test oracle. In the future, we will use more seed programs and ingredients to further reduce these threats.

Table 6: Comparison among JavaTailor, VECT, and VECT's variants

Method	VECT	VECT$_{noCluster}$	VECT$_{noFeedback}$	VECT$_{noChecksum}$	JavaTailor
#Unique Inconsistencies	45	30	34	17	12

The *construct* threats to validity mainly lie in randomness and the method of identifying duplicate inconsistencies detected by normal execution outputs. To reduce the threat of randomness, we conducted five differential-testing experiments instead of repeating one experiment several times. Indeed, we obtained the consistent conclusions from the five experiments. Regarding the latter threat, we have discussed it in Section 4.3.

6 RELATED WORK

JVM Testing. Our work is the most related to JVM testing. Besides the compared technique (i.e., JavaTailor) in our study, there are some other JVM testing techniques [10, 18, 42, 55, 56]. For example, Chen et al. proposed two mutation-based techniques, i.e., classfuzz [18] and classming [17], which design some mutation rules (e.g., changing the modifier or type of a variable) to minorly change the given seed program for generating a new test program. Zhang et al. [56] proposed *JAttack*, which generates test programs by filling holes in the template classes with randomly generated expressions and values. Hwang et al. [30] proposed *JUSTGen*, which designs a domain-specific language for the JNI specification and identifies unspecified cases in the specification to generate test programs.

Different from them, our work proposes Vectorized JVM Testing to promote the performance of synthesis-based JVM testing. Its key insight is to improve the ingredient exploration by vectorizing ingredients for clustering and guiding the ingredient selection based on the selection and testing history.

Compiler Testing. Our work is also related to compiler testing [12, 15, 16, 45], since both of them take programs as inputs. Here, we briefly introduce the related work on compiler testing [14]. For example, Yang et al. [54] proposed *Csmith*, which is a grammar-based C program generator. Lidbury et al. [36] proposed *CLsmith* based on Csmith, which designs six modes to generate test programs for OpenCL compilers. Le et al. [33] proposed Equivalence Modulo Inputs (EMI) for testing C compilers, which constructs equivalent programs under a set of inputs through program mutation. Windsor et al. [51] proposed *C4*, which generates multi-thread C programs and the corresponding post-conditions to test the concurrency behaviours of C compilers. Donaldson et al. [21] proposed *GLFuzz*, which designs semantics-preserving program transformations to generate test programs for shader compilers. Holler et al. [29] proposed a synthesis-based JS program generation technique, but like JavaTailor, it randomly combines code fragments by treating them equally and individually.

Different from them, our work targets JVM testing by vectorizing ingredients (via code representation) to improve synthesis-based JVM testing. This idea is also novel in compiler testing, and in the future we can extend VECT to this area.

Code Representation. VECT borrows the power from the area of code representation. In recent years, code representation has been used to solve some software engineering tasks, e.g., code search [9, 35], API recommendation [31, 52], code clone detection [24, 50], and program repair [39, 53]. Different from them, our work is the first to incorporate code representation to JVM testing (i.e., encoding ingredient semantics for improving synthesis-based JVM testing). Besides our studied code representation models, there are some other models that were pre-trained and released, e.g., CoTexT [40] and GraphCodeBERT [27]. In the future, we can evaluate their effectiveness in our task to improve the performance of VECT.

7 CONCLUSION

To promote the performance of synthesis-based JVM testing, in this work, we propose a novel technique (called VECT) by vectorizing program ingredients. It aims to reduce the huge ingredient space by clustering semantically similar ingredients based on semantic vectors. Then, VECT designs a feedback-driven ingredient selection strategy to more efficiently explore the reduced space, and enhances the existing test oracle by monitoring the results of various intermediate variables in the synthesized test program to more sufficiently capture the triggered bugs. The results on three popular JVMs (i.e., HotSpot, OpenJ9, and Bisheng JDK) demonstrate the superiority of VECT over the state-of-the-art JavaTailor. In particular, 15 of 26 previously unknown bugs detected by VECT have been confirmed/fixed by developers.

ACKNOWLEDGMENTS

We thank all the ISSTA anonymous reviewers for their valuable comments. We also thank all the JVM developers for analyzing and replying to our reported bugs. The work has been supported by the National Natural Science Foundation of China under Nos. 62232001 and 62002256. This work is also sponsored by CCF-Huawei Populus Grove Fund.

REFERENCES

[1] 2022. Bisheng. https://www.openeuler.org/zh/other/projects/bishengjdk.
[2] 2022. Gcov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[3] 2022. GIJ. https://web.archive.org/web/20070509055923/http://gcc.gnu.org/java.
[4] 2022. HotSpot. http://openjdk.java.net.
[5] 2022. OpenJ9. https://www.eclipse.org/openj9.
[6] 2022. scikit-learn. https://scikit-learn.org/stable/.
[7] 2022. VECT. https://github.com/gaotravor/VECT
[8] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).
[9] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* (2019).
[10] Inc. Azul Systems. 2018. AzulSystems/JavaFuzzer: Java* Fuzzer for Android*. https://github.com/AzulSystems/JavaFuzzer
[11] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Infercode: Self-supervised learning of code representations by predicting subtrees. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1186–1197.
[12] 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*.
[13] 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*.

[14] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* (2020).

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

[16] 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.* (2021).

[17] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1257–1268.

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

[19] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. 2020. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555* (2020).

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[21] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.* (2017).

[22] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *kdd*.

[23] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[24] Yi Gao, Zan Wang, Shuang Liu, Lin Yang, Wei Sang, and Yuanfang Cai. 2019. TECCD: A Tree Embedding Approach for Code Clone Detection. In *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*.

[25] Vincenzo Gervasi and Roozbeh Farahbod. 2009. JASMine: Accessing java code from CoreASM. In *Rigorous Methods for Software Construction and Analysis*.

[26] Matthias Grimmer, Manuel Rigger, Roland Schatz, Lukas Stadler, and Hanspeter Mössenböck. 2014. Trufflec: Dynamic execution of c on a java virtual machine. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*.

[27] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[28] John A Hartigan and Manchek A Wong. 1979. Algorithm AS 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)* (1979).

[29] Christian Holler, Kim Herzig, Andreas Zeller, et al. 2012. Fuzzing with Code Fragments.. In *USENIX Security Symposium*.

[30] 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, ICSE 2021, Madrid, Spain, 22-30 May 2021*.

[31] Yuning Kang, Zan Wang, Hongyu Zhang, Junjie Chen, and Hanmo You. 2021. APIRecX: Cross-Library API Recommendation via Pre-Trained Language Model. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.).

[32] Leonard Kaufman and Peter J Rousseeuw. 2009. *Finding groups in data: an introduction to cluster analysis*.

[33] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* (2014).

[34] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461* (2019).

[35] Zongjie Li, Pingchuan Ma, Huaijin Wang, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2022. Unleashing the Power of Compiler Intermediate Representation to Enhance Neural Program Embeddings. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*.

[36] 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, Portland, OR, USA, June 15-17, 2015*.

[37] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[38] Lili Mou, Ge Li, Zhi Jin, Lu Zhang, and Tao Wang. 2014. TBCNN: A tree-based convolutional neural network for programming language processing. *arXiv preprint arXiv:1409.5718* (2014).

[39] Vinicius Paulo L. Oliveira, Eduardo Faria de Souza, Claire Le Goues, and Celso G. Camilo-Junior. 2018. Improved representation and genetic operators for linear genetic programming for automated program repair. *Empir. Softw. Eng.* (2018).

[40] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Anibal, Alec Peltekian, and Yanfang Ye. 2021. Cotext: Multi-task learning with code-text transformer. *arXiv preprint arXiv:2105.08645* (2021).

[41] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.* (2020).

[42] Inc. Red Hat. 2018. shipilev/JavaFuzzer: Java* Fuzzer for JVM. https://github.com/shipilev/JavaFuzzer

[43] Peter J Rousseeuw. 1987. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics* (1987).

[44] Ketan Rajshekhar Shahapure and Charles Nicholas. 2020. Cluster quality analysis using silhouette score. In *2020 IEEE 7th International Conference on Data Science and Advanced Analytics (DSAA)*.

[45] 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*.

[46] Jim Smith and Ravi Nair. 2005. *Virtual machines: versatile platforms for systems and processes*. Elsevier.

[47] Suvrit Sra, Sebastian Nowozin, and Stephen J Wright. 2012. *Optimization for machine learning*.

[48] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F Bissyandé. 2020. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[49] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[50] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*.

[51] Matt Windsor, Alastair F. Donaldson, and John Wickerson. 2021. C4: the C compiler concurrency checker. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*.

[52] Rensong Xie, Xianglong Kong, Lulu Wang, Ying Zhou, and Bixin Li. 2019. HiRec: API Recommendation using Hierarchical Context. In *30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019, Berlin, Germany, October 28-31, 2019*.

[53] Chen Yang. 2021. Accelerating redundancy-based program repair via code representation learning and adaptive patch filtering. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*.

[54] 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.

[55] Takahide Yoshikawa, Kouya Shimura, and Toshihiro Ozawa. 2003. Random Program Generator for Java JIT Compiler Test System. In *3rd International Conference on Quality Software (QSIC 2003), 6-7 November 2003, Dallas, TX, USA*.

[56] Zhiqiang Zang, Nathan Wiatrek, Milos Gligoric, and August Shi. 2022. Compiler Testing using Template Java Programs. *CoRR* (2022).

[57] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 39–51.

[58] Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-driven test program synthesis for JVM testing. In *Proceedings of the 44th International Conference on Software Engineering*. 1133–1144.