Selecting Initial Seeds for Better JVM Fuzzing

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

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

Dong Wang
College of Intelligence and Computing
Tianjin University
Tianjin, China
dong_w@tju.edu.cn

Yile Guo
College of Intelligence and Computing
Tianjin University
Tianjin, China
gyl_666@tju.edu.cn

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

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

Abstract—**JVM fuzzing techniques serve as a cornerstone for guaranteeing the quality of implementations. In typical fuzzing workflows, initial seeds are crucial as they form the basis of the process. Literature in traditional program fuzzing has confirmed that effectiveness is largely impacted by redundancy among initial seeds, thereby proposing a series of seed selection methods. JVM fuzzing, compared to traditional ones, presents unique characteristics, including large-scale and intricate code, and programs with both syntactic and semantic features. However, it remains unclear whether the existing initial seed selection methods are suitable for JVM fuzzing and whether utilizing program features can enhance effectiveness. To address this, we devise a total of 10 initial seed selection methods, comprising coverage-based, prefuzz-based, and program-feature-based methods. We then conduct an empirical study on three JVM implementations to extensively evaluate the performance of the initial seed selection methods within two state-of-the-art fuzzing techniques (JavaTailor and VECT). Specifically, we examine performance from three aspects: (i) effectiveness and efficiency using widely studied initial seeds, (ii) effectiveness using the programs in the wild, and (iii) the ability to detect new bugs. Evaluation results first show that the program-feature-based method that utilizes the control flow graph not only has a significantly lower time overhead (i.e., 30s), but also outperforms other methods, achieving 142% to 269% improvement compared to the full set of initial seeds. Second, results reveal that the initial seed selection greatly improves the quality of wild programs and exhibits complementary effectiveness by detecting new behaviors. Third, results demonstrate that given the same testing period, initial seed selection improves the JVM fuzzing techniques by detecting more unknown bugs. Particularly, 21 out of the 25 detected bugs have been confirmed or fixed by developers. This work takes the first look at initial seed selection in JVM fuzzing, confirming its importance in fuzzing effectiveness and efficiency.**

Index Terms—**Java Virtual Machine, JVM Fuzzing, Initial Seed Selection, Empirical Study**

I. INTRODUCTION

The Java Virtual Machine (JVM) serves as a critical infrastructure for the Java platform, providing a consistent runtime environment that allows developers to create and deploy Java applications across diverse hardware and operating systems without needing modification. Given its foundational role, numerous JVM fuzzing techniques have been developed to ensure the quality of JVM implementations [1]–[3]. These techniques typically follow a general workflow, which involves (1) preparing a set of initial seeds, (2) generating test programs through mutations or code synthesis applied to these seeds, and (3) testing JVMs with the generated test programs and providing feedback for subsequent iterations. Among these steps, the initial seeds form the basis of the entire fuzzing process and thus play a critical role in determining the overall effectiveness of the fuzzing campaign.

In practice, fuzzing budgets are often constrained, and redundancy among initial seeds, concerning the behaviors they induce during the fuzzing process, is a common occurrence [4]. This redundancy necessitates spending a significant amount of time on initial seeds with overlapping behaviors for fuzzing, consequently limiting overall effectiveness within the allocated time budget [5]. Several studies have been conducted to confirm such an influence on fuzzing *traditional programs*, and proposed some methods of selecting a subset of initial seeds for improving the fuzzing effectiveness, such as coverage-based [5], [6] and prefuzz-based [4] methods.

JVM fuzzing presents unique characteristics distinct from traditional program fuzzing. On the one hand, JVM code tends to be large-scale and intricate [7], resulting in significant overhead in collecting coverage and executing test programs. Therefore, both coverage-based and prefuzz-based methods suffer from cost issues. On the other hand, JVM fuzzing operates on programs containing rich syntactic and semantic features, and several existing works have highlighted the correlation between test program features and JVM fuzzing effectiveness [3], [8]. Given these unique characteristics, *it remains uncertain whether existing initial seed selection methods are suitable for JVM fuzzing and whether incorporating test program features can enhance initial seed selection effectiveness*. To address these open questions, we conducted an empirical study to investigate the influence of initial seed

---

* Junjie Chen is the corresponding author.

selection in the context of JVM fuzzing, with the aim of further improving the effectiveness of JVM fuzzing within limited time budgets.

In this study, we investigated a total of 10 initial seed selection methods, which comprised two coverage-based methods (**CISS**) [4], [6], one prefuzz-based method (**PISS**) [4], and seven program-feature-based methods (**FISS**). The coverage-based and prefuzz-based methods were originally proposed in the context of traditional program fuzzing, and we adopted them for JVM fuzzing. Specifically, they utilize test coverage and short-time fuzzing results as metrics to assess the fuzzing capability of each initial seed for selection, respectively. The remaining seven methods were particularly designed by us for JVM fuzzing, considering different ways of utilizing program features. Specifically, these methods utilize textual features, Abstract Syntax Tree (AST) features, Control Flow Graph (CFG) features, and code semantics obtained from various code representation models to measure the diversity of initial seeds for selection, respectively. Further details of these studied methods will be presented in Section III.

Based on the set of methods, we performed an experiment to address the following three research questions.

(**RQ1**): How do these initial seed selection methods perform in JVM fuzzing? We first seek to investigate the performance of the studied initial seed selection methods in terms of effectiveness and efficiency. In particular, we used three widely-used JVM implementations (i.e., HotSpot [9], OpenJ9 [10], and Bisheng JDK [11]) as subjects, two widely-used sets of initial seeds as the ones for selection, and two state-of-the-art JVM fuzzing techniques (i.e., JavaTailor [2] and VECT [3]) for evaluating each selected subset of initial seeds. To make our conclusions have statistical significance, we used the historical versions of these JVM implementations as the studied JVM fuzzing techniques can detect more bugs in them.

Results: The program-feature-based method that utilizes CFG features, FISS_{CFG}, not only has a significantly lower time overhead (i.e., 30s), but also outperforms other methods, achieving 142% to 269% improvement compared to the full set of initial seeds. Specifically, we recommend using FISS_{CFG} with a 50% budget for initial seed selection, which can optimize performance and resource utilization in JVM fuzzing.

(**RQ2**): How does initial seed selection perform on programs in the wild for improving JVM fuzzing? Intuitively, any Java programs in the wild can be used as initial seeds for JVM fuzzing. It is plausible that some of these programs may introduce new behaviors compared to widely-studied initial seeds, thereby potentially improving JVM fuzzing. However, there is also a substantial number of programs that may not introduce new aspects for JVM fuzzing. Randomly collecting a large number of programs in the wild as initial seeds is unlikely to significantly enhance the effectiveness of JVM fuzzing within limited time budgets. This may be a primary reason why existing JVM fuzzing techniques do not use them as initial seeds. However, the existence of initial seed selection methods presents an opportunity to strategically incorporate programs in the wild. This may help leverage the benefits

of such programs effectively and thus improve JVM testing. Therefore, we designed RQ2 in our study, which not only investigates the generalizability of our conclusions but also explores the feasibility of a new aspect to enhance the effectiveness of JVM fuzzing. Specifically, we collected a set of programs in the wild as a new corpus and then repeated the first experiment on it.

Results: Programs in the wild complement the widely-studied initial seeds by detecting new JVM behaviors. The subset of wild programs selected by FISS_{CFG} boosts detection capability, achieving an improvement of 8.3% to 108% compared to the full set of initial seeds. Moreover, results underscore the crucial role of initial seed selection in enhancing the effectiveness of wild programs, and also reinforce the generalizability of the top-performing method, FISS_{CFG}.

(**RQ3**): Can initial seed selection survive existing JVM fuzzing techniques for detecting new bugs? Based on RQ1 and RQ2 results, it is evident that initial seed selection is helpful in enhancing the effectiveness of JVM fuzzing by allocating more time to diverse (selected) seeds and enabling the effective incorporation of programs in the wild. However, the ultimate aim of fuzzing is to find bugs in JVM, thus we further investigated whether initial seed selection also helps improve studied fuzzing techniques in detecting previously unknown bugs. Specifically, we applied each subset of initial seeds selected by different seed selection methods on VECT to test the latest versions of three JVM implementations. We then compared the number of unknown bugs detected by each method (including the full set) within the same testing period.

Results: Initial seed selection FISS_{CFG} improves the JVM fuzzing techniques by detecting more previously unknown bugs within the same testing period. Of the 25 submitted bugs, the developers have confirmed and fixed 21 bugs, and 7 can only be detected using initial seeds from wild programs.

To sum up, our major contributions are four-fold:

- Design of seven initial seed selection methods specifically tailored for JVM fuzzing, leveraging various features in test programs.
- Implementation of all studied initial seed selection methods as an open-source toolkit [12], facilitating practical use and future research in the field.
- Conducting an extensive study to investigate the impact of initial seed selection on the effectiveness of JVM fuzzing, addressing three research questions (RQs).
- Obtaining a series of findings and implications for improving JVM fuzzing.

II. BACKGROUND AND RELATED WORK

A. JVM Fuzzing

Fuzzing is one of the most popular and effective methods to find bugs and vulnerabilities in software [7], [13]–[15]. In the context of JVM, diverse fuzzing techniques have been proposed to ensure the quality of JVM implementations. The majority of them are seed-driven, such as the state-of-the-art *JavaTailor* and *VECT* that are experimented in this study.

- **JavaTailor** adopts a history-driven approach for synthesizing test programs. It designs five types of ingredients, extracted from bug-revealing test programs. These ingredients are then inserted into seed programs, and syntactic and semantic constraints within the constituents are automatically rectified to produce valid synthesized programs for differential testing.
- **VECT** streamlines the extensive ingredient space by vectorizing program ingredients through advanced code representation. It then employs a feedback-based ingredient selection strategy to enhance the effectiveness of test program generation. Furthermore, VECT incorporates an enhanced test oracle to broaden the bug detection capability of current JVM testing.

Additionally, there are other techniques that are also seed-driven. For example, Chen et al. introduced *classfuzz* [16] and *classming* [1], which mutate bytecode and control flow to generate test programs with broader coverage, respectively. Wu et al. proposed *JITfuzz* [17], which designs mutation operators related to JIT defects to generate test programs. *SJFuzz* [18] optimizes the scheduling of seed programs and mutation operators to enhance the effectiveness of classming.

Apart from the seed-driven techniques, some efforts target generating test programs by starting from an empty program or a program with holes [19]–[21]. For instance, Zhang et al. proposed *JAttack* [21], which fills holes in template classes with randomly generated expressions and values to generate test programs. Hwang et al. introduced *JUSTGen* [20], which uses JNI specifications to identify unspecified scenarios for generating corner cases.

Our work is applicable to various JVM fuzzing techniques that require initial seeds. We devised a set of methods for selecting initial seeds, in order to extensively study their effect on fuzzing capability.

B. Initial Seed Selection

In seed-driven fuzzing workflows, the selection of initial seeds is typically one of the first steps and plays a significant role in the overall effectiveness of fuzzing. Several studies have highlighted the importance of high-quality seeds on fuzzers' performance. For instance, Klees et al. [22] asserted that fuzzer's performance can greatly vary on the same program, depending on the seed used. Herrera et al. [5] evaluated how seed selection affects a fuzzer's ability and their findings suggest that seed selection is a critical step that must be considered prior to launching any fuzzing campaign. Remarkably, practitioners (such as the developers of the Mozilla Firefox browser [23]) also recognized the importance of seed selection.

Several methods have been proposed to select a subset of initial seeds, with the aim of improving the quality of initial seeds and reducing the budget of fuzzing. Specifically, a common method in traditional software testing is the corpus minimization technique, which selects the smallest subset of the corpus that maintains the equivalent coverage to the entire corpus. This technique involves widely-adopted coverage-based methods, such as *PeachSet* [6], *MinSet* [4], and *OptiMin* [5]. For example, MinSet, an unweighted greedy-reduced minimization, sorts seed files based on their coverage

Table I: Studied initial seed selection methods

Type	Method	Input	Strategy
White-box	$CISS_P$	Coverage	Greedy
	$CISS_M$	Coverage	Greedy
Black-box	PISS	Fuzzing Result	Greedy
	$FISS_{TS}$	Token	FPS
	$FISS_{AST}$	AST	FPS
	$FISS_{CFG}$	CFG	FPS
	$FISS_{CB}$	Token	FPS
	$FISS_{IC}$	AST	FPS
	$FISS_{PB}$	Token	FPS
	$FISS_{CT}$	Token	FPS

increments and chooses the one with the maximum increase. The prior work reported that PeachSet found the highest number of bugs and MinSet performed the best in terms of the minimization ability [4]. The prefuzz-based method *HotSet* [4] executes each seed file for an equal amount of time and sorts them based on the number of known defects each file uncovers.

Despite extensive study of initial seed selection in traditional software, its role in JVM fuzzing remains largely unexplored. To address this, we conducted the first study to investigate the effect of various selection methods on JVM fuzzing performance. Different from those methods aiming to minimize the initial seeds, as the initial step, our focus is to understand the effect of selecting subsets of the seeds. Furthermore, certain existing methods (such as HotSet) are unable to achieve minimization. Hence, to make a fair comparison among these methods, we opted to establish a selected subset of initial seeds. Current selection methods may cause considerable overhead. The code space of JVM is vast, and collecting coverage for each seed program takes about 500 seconds. The time required to apply existing coverage-based methods to a large corpus is substantial. Similarly, techniques based on known bugs require individual fuzzing for each seed program, which further consumes significant time resources. To mitigate this challenge, we novelly designed several program-feature-based methods and evaluated their performance.

III. STUDIED INITIAL SEED SELECTION METHODS

Drawing inspiration from existing works in the realm of traditional software, we devised a set of initial seed selection methods tailored for JVM testing. Table I presents a summary of the studied ten methods including their types, names, inputs, and search strategies. These methods are categorized into three groups: coverage-based, prefuzz-based, and program-feature-based methods. For simplicity, we will use the term corpus to refer to the entire set of initial seeds in the following sections.

A. Coverage-based Method

Two white-box initial seed selection methods based on coverage metrics are introduced: $CISS_P$ and $CISS_M$. Like PeachSet and MinSet, $CISS_P$ sorts seed programs based on coverage metrics, then selects the top-k percent as the corpus

subset. On the other hand, CISS$_M$ sorts these programs based on coverage increments and stops the selection process once all candidate seed programs show zero coverage increments. To accommodate our experiment implementation, when the selection process stops, CISS$_M$ transitions into a mode the same as CISS$_P$. In this mode, candidate seed programs are sorted based on coverage metrics. Likewise, the top-k percent of these programs is selected as the corpus subset. Note that when several seed programs have the same coverage, we randomly select one of them to break the tie following the existing work [4], [24].

B. Prefuzz-based Method

PISS is a black-box method adapted from HotSet. Specifically, PISS performs fuzzing on each seed program for t seconds, recording the number of bugs each seed program uncovers. It then selects the top-k percent with the highest number of bugs to form the corpus subset. Following the existing work [4], we conduct fuzzing on each seed program for 5 minutes (t = 300) to compute the subset of initial seeds selected by PISS in our experiments.

C. Program-Feature-based Method

Previous work has emphasized the correlation between test program features and JVM fuzzing effectiveness [25], [26]. Specifically, similar program features may possess the ability to trigger similar bugs or the absence of bugs. Moreover, the triggering of bugs is frequently associated with specific program features rather than common ones. Hence, we further propose a program-feature-based initial seed selection method (*FISS*) by extracting program features for evaluation. Based on whether we extract program features using statistical models or pre-trained models, we categorize them into traditional features and code semantics. Below we describe the process for extracting program features including traditional features and code semantics, as well as the corpus subset selection.

Traditional feature extraction. Three commonly utilized types of information in program analysis are selected: textual information, syntactic information, and semantic information [27]–[29]. Specifically, we analyze the Token Sequence (TS), Abstract Syntax Tree (AST), and Control Flow Graph (CFG) of programs to extract traditional features, respectively.

- *TS features*: Each token in the corpus is treated as a dimension in the feature vector, and the TF-IDF [30] is used to compute the score of each token in the seed program. Tokens that are absent receive a value of 0.
- *AST features*: The Java syntax analysis tool JDT [31] is utilized to obtain the AST of each seed program and extract tree-based n-gram chains. Existing work [32] has shown that setting n to 3 yields better representation performance, so we also extract 3-gram chains for our method. Each 3-gram chain in the corpus is regarded as a dimension in the feature vector, and the frequency of occurrence of each 3-gram chain in the seed program is counted. Chains that do not exist receive a value of 0.

- *CFG features*: The Java bytecode analysis tool Soot [33] is used to obtain the CFG and extract graph-based 3-gram chains. Each node in the CFG represents a Jimple instruction since Soot analyzes bytecode using the intermediate language, Jimple. Similar to AST features, we count the frequency of each 3-gram chain in the seed program as a dimension in the feature vector.

As shown in Table I, the methods for initial seed selection using the above three features are referred to as FISS$_{TS}$, FISS$_{AST}$, and FISS$_{CFG}$.

Code semantic extraction. Recent studies have demonstrated the efficacy of leveraging pre-trained code representation models in various software engineering tasks [34]–[37]. Therefore, employing pre-trained models to obtain program feature vectors is intuitive. Particularly, in the study of JVM testing [3], the code representation models (CodeBERT [38], InferCode [39], codeT5 [40], and PLBART [41]) play a pivotal role in semantic vectorization. Encouraged by this, we also adopt these four pre-trained models, aiming to explore their effectiveness in code representation for initial seed selection tasks. To do so, we use open-source pre-trained models directly for zero-shot inference of the code representation vectors from each seed program. If the seed program is too long, it is then divided into slices, and the code representation vectors for each slice are averaged. As shown in Table I, the methods for initial seed selection using features extracted from these four pre-trained models are referred to as FISS$_{CB}$, FISS$_{IC}$, FISS$_{CT}$, and FISS$_{PB}$.

Corpus subset selection. After feature extraction, FISS sorts all seed programs and selects the top-k percent to form the corpus subset. The greedy algorithm is applicable to sortable data, e.g., coverage information used by CISS and the number of detected bugs used by PISS. However, FISS represents program features as vectors for selection, which are unsortable, so the greedy strategy is not applicable. Some techniques built on the concept of adaptive random testing (ART) [42]–[44] argue that uniformly distributed test cases are more likely to detect bugs with fewer test cases than ordinary random testing. Furthest Point Sampling [45] (FPS) is widely used in existing work to select uniformly distributed test cases [46], [47]. Following this, we employ FPS to sort the seed programs in the corpus by measuring distance between vectors, as seed programs with similar program features may reveal similar bugs. The central concept of FPS is to prioritize selecting points that are farthest away from the centroid of the already selected points. This strategy ensures that the selected points exhibit distinct features, enabling the prioritization of more unique points in the process.

Give a set of initial seeds denoted as $\mathcal{S} = \{s_1, s_2, .., s_n\}$ (n refers to the number of seeds) and the budget of initial seed selection denoted as k, the target of FISS is to select k percent of the seed programs from $\mathcal{S}$ as a corpus subset. Specifically, FISS first needs to extract the feature vectors for each seed program, denoted as $\mathcal{V} = \{v_1, v_2, ..., v_n\}$ and $v_i = f(s_i)$, where f refers to the feature extraction method and $f(s_i)$ refers to the feature vector of s_i. Then, FISS sorts the seed

programs using FPS. To ensure stability, we first compute the feature centroid of all seed programs as shown in Formula 1.

$$C = (c_1, c_2, ..., c_m) = \frac{1}{n} \sum_{i=1}^{n} f(s_i) \qquad (1)$$

Where c_i refers to one dimension of a feature vector, m refers to the number of dimensions. Then, FISS selects the point farthest from the centroid as the first seed program shown as in Formula 2.

$$t_1 = \arg \max_{s_i \in \mathcal{S}} dist(f(s_i), C) \qquad (2)$$

Where t_1 refers to the first selected point and $dist(v_i, C)$ calculates the Euclidean distance between feature vectors. We use $\mathcal{T}$ to represent the set of seeds that have been selected. For each seed $s_i \in \mathcal{S} - \mathcal{T}$, we need to compute its minimum distance from the selected seeds in the set $\mathcal{T}$ and select the seed with the maximum distance shown as Formula 3.

$$t_{next} = \arg \max_{s_i \in \mathcal{S} - \mathcal{T}} \min_{t_j \in \mathcal{T}} dist(f(s_i), f(t_j)) \qquad (3)$$

This iterative process continues until all seed programs are included in $\mathcal{T}$, indicating the completion of sorting. Lastly, we exclusively select the top-k percent of seed programs from $\mathcal{T}$ to compose the corpus subset.

IV. EVALUATION DESIGN

A. Studied JVMs

In line with existing research focusing on JVM testing [3], we studied three popular JVMs: Hotspot, OpenJ9, and Bisheng JDK. Table II shows the summary of studied JVMs with their versions. Unlike existing work [2], [3], we did not use OpenJDK12-14 as they are no longer maintained. Alternatively, we added OpenJDK17 as a subject. Note that, for OpenJDK-8, each JVM includes both an older build and the latest build. This is because relatively older JVM builds often contain more bugs, which allows for a better evaluation of the fuzzer's effectiveness. By comparing the outputs of older JVM builds to those of the latest builds, we can calculate the evaluation metric of unique inconsistencies. Similarly, by examining the outputs of the latest builds, we can calculate the evaluation metric of unknown bugs. Further details regarding these evaluation metrics are provided in Section IV-D.

B. Studied Corpus

Two types of corpus are studied: benchmark corpus and open-source corpus. Table III provides basic information about these corpus, where *#Size* represents the number of seed programs and *#Inst* represents the number of Jimple instructions.

Benchmark Corpus. To ensure fair evaluation, we used the same corpus as the prior works focusing on JVM testing [2], [3]. Specifically, we filtered out those corpus that contained only one seed program, as they did not require seed selection. Finally, we obtained two corpus that contained multiple seed programs (i.e., P1 and P2) for this study. The studied two corpus consist of historical bug-revealing test programs collected

Table II: Studied JVM versions

OpenJDK Version	JVM Implementation	Version
OpenJDK8	HotSpot	build 25.0-b70
		build 25.402-b06
	OpenJ9	build openj9-0.8.0
		build openj9-0.43.0
	Bisheng JDK	build 25.302-b13
		build 25.392-b12
OpenJDK11	HotSpot	build 11.0.22+7
	OpenJ9	build openj9-0.43.0
	Bisheng JDK	build 11.0.21+12
OpenJDK17	HotSpot	build 17.0.10+7
	OpenJ9	build openj9-0.43.0
	Bisheng JDK	build 17.0.9+12

Shadow represents the used old build of the corresponding JVM.

Table III: Studied corpus

ID	ProjectName	#Size	#Inst	Source
P1	HotSpot-tests	469	104,797	Hotspot
P2	OpenJ9-tests	586	105,212	OpenJ9
P3	CollectProject	942	127,798	GitHub

from the Hotspot and OpenJ9 repositories [2], containing 469 and 586 test programs, respectively.

Open-Source Corpus. To enhance the corpus diversity for JVM testing, we propose a novel corpus sourced from the open-source community on GitHub, containing a larger set of test programs. Figure 1 provides an overview of the corpus collection process. The process consists of three phases: search, compilation, and differential testing.

(1) During the search phase, we collected URLs of projects from GitHub using the following criteria: the projects are written in Java, they have more than 100 stars, and the project structure is Maven. Specifically, we utilized the GitHub API [48] as our search tool and set the aforementioned criteria to retrieve repository URLs. Because the GitHub API limits the return to fewer than 1,000 repositories, we segmented the star counts to ensure that the number of repositories returned each time stays within this limit. After the search phase, we obtained a total of 6,507 repository URLs.

(2) During the compilation phase, we first filtered out those repositories that either do not contain a *pom.xml* file or lack a Main entry. The presence of a *pom.xml* file facilitates the automation of the compilation process, while the Main entry point is essential for obtaining seed programs. We successfully cloned 460 of the surviving repositories locally for
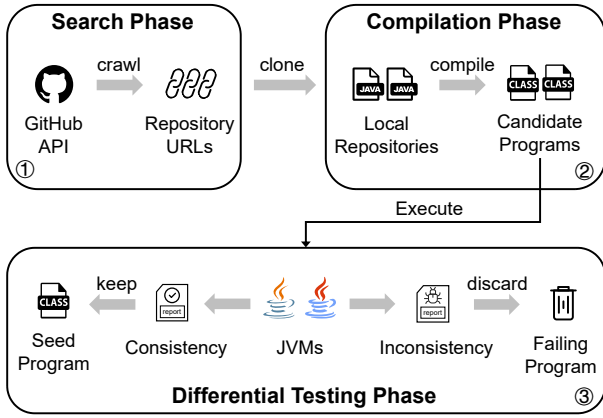
Figure 1: Automated collection of open-source corpus

compilation. Specifically, we used the *mvn package* command to compile, however, if repositories lacked dependencies and could not execute the command, we resorted to using *javac* to compile the Main entry in the repository. This approach ensures that we fully utilize each repository, maximizing the potential for successful compilation. After the compilation phase, we filtered out 256 repositories that either could not be compiled or did not have a successfully complied Main entry. Finally, 204 repositories and 1,173 Main entries remained.

③ During the differential testing phase, we employed older JVM builds for testing. The specific versions of them can be found in our replication package. One issue with older JVMs is that certain seed programs may directly trigger inconsistencies, which could potentially interfere with the effectiveness of evaluations. To mitigate this issue, we performed an interference on the Main entry in each repository to verify whether any of the seed programs were executing properly. This extra step aids in ensuring that the seed programs are functioning as expected, enabling us to identify and address any discrepancies before evaluations. After the differential testing phase, we filtered out 50 repositories that did not obtain any consistent Main entry, leaving us with 154 repositories. In particular, we filtered out 231 inconsistent Main entry classes. As shown in Table III, our proposed corpus (referred to as P3) finally consists of 942 seed programs and 127,798 Jimple instructions.

C. Experimented Fuzzers

In this study, two seed-driven JVM fuzzers are selected: *JavaTailor* and *VECT*. These two fuzzers construct 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. They are state-of-the-art and have outperformed other widely-studied JVM testing techniques as demonstrated by prior work [2], [3]. The detailed descriptions of these two fuzzers are presented in Section II.

Note that we opted not to use the enhanced test oracle featured in VECT. This choice was made because we did not manually filter or modify the open-source seed programs. Certain variables, like randomness and timestamps, can introduce non-determinism. Therefore, the Correcting Commit [49] used

for inconsistency deduplication may potentially introduce false positives, which could impact our results. Therefore, this study employed the same test oracle as JavaTailor in the evaluation. Considering the effectiveness of VECT, we adopted PLBART, as recommended in VECT, for semantic vectorization.

D. Measurement

Time overhead: To compare the time overheads of each initial seed selection method, we recorded the time required for three key phases: data collection, data processing, and corpus subset selection. Specifically, during the data collection phase, CISS needs to collect coverage data from the JVM and PISS gathers the fuzzing results of each seed program. On the other hand, FISS directly utilizes program features, thus eliminating the need for data collection. During the data processing phase, CISS constructs bitmaps to facilitate subsequent operations, PISS performs deduplication and counts the number of inconsistencies detected, and FISS extracts program features and represents them as vectors for further analysis. During the corpus subset selection phase, either greedy strategies or Farthest Point Sampling (FPS) are employed for sorting, and the top-k percent of seed programs are selected.

Number of unique inconsistencies: In the differential testing experiments conducted on relatively older JVM builds, each studied method is likely to detect some inconsistencies (i.e., potential real bugs, false positives, and duplicates) during the same testing time. Therefore, we executed another round of differential testing employing the latest JVM builds to verify whether the inconsistencies remain. If the inconsistency disappears, it suggests that a known bug has been fixed; otherwise, we performed manual analysis to determine whether the inconsistency is due to a potential bug. Additionally, some inconsistencies may be duplicated due to the same bug, hence we further de-duplicated them based on the crash messages, which is the most commonly used automatic method in existing work [3]. We used the number of inconsistencies after de-duplication as the number of unique inconsistencies.

Number of previously unknown bugs: In the differential-testing experiments conducted on newer JVM builds, each studied technique is likely to detect some inconsistencies. In RQ3, we chose the more efficient VECT to conduct differential testing experiments on the latest versions of JVM. Existing research demonstrates that while VECT and JavaTailor share the same exploration space, VECT exhibits comparatively greater exploration efficiency [3]. To verify the authenticity of unknown bugs, we manually analyzed each inconsistency to determine whether a discrepancy is a real bug. Then, we created bug reports for test programs and submitted them to the corresponding repositories. The number of unknown bugs is measured based on feedback from developers.

E. Implementation and Environment

We implemented all initial seed selection methods in Java. The extraction and analysis of AST and CFG relied on APIs provided by JDT and Soot. We utilized the pre-trained code representation models from existing work [3], using their

Table IV: Comparison results of different methods in terms of the number of unique inconsistencies

		JavaTailor			VECT		
	FullSet	5.4			9.0		
		20%	35%	50%	20%	35%	50%
	RandomSet	5.0	4.6	5.4	5.6	6.2	7.4
P1	$CISS_M$	**7.4**	**6.0**	**7.8**	7.2	**9.4**	8.4
	$CISS_P$	5.0	5.2	**6.4**	5.4	6.6	7.4
	PISS	**6.4**	**7.2**	**8.0**	**9.4**	8.6	7.8
	$FISS_{TS}$	**7.2**	**8.4**	**9.6**	6.6	7.6	7.6
	$FISS_{AST}$	**8.6**	**9.0**	**9.4**	8.6	8.6	**10.0**
	$FISS_{CFG}$	**12.6**	**13.0**	**12.0**	**12.8**	**14.8**	**16.2**
	$FISS_{CB}$	**7.6**	**7.0**	**8.6**	6.2	7.4	8.2
	$FISS_{IC}$	**7.2**	**8.8**	**9.4**	7.8	7.8	8.4
	$FISS_{PB}$	**7.4**	**8.2**	**8.0**	7.6	7.8	7.2
	$FISS_{CT}$	**8.0**	**8.0**	**9.0**	7.2	8.6	8.2
	FullSet	6.4			7.8		
		20%	35%	50%	20%	35%	50%
	RandomSet	6.2	**7.4**	5.8	4.0	5.0	5.4
P2	$CISS_M$	**8.6**	**8.2**	**8.6**	**9.2**	**8.6**	**9.2**
	$CISS_P$	**9.2**	**7.4**	**7.8**	**9.4**	**9.4**	**9.2**
	PISS	**7.8**	**7.2**	**7.8**	**9.6**	**8.0**	7.4
	$FISS_{TS}$	**7.0**	**7.2**	**6.8**	6.4	6.0	5.6
	$FISS_{AST}$	**11.8**	**9.2**	**11.6**	**9.0**	**9.4**	**10.2**
	$FISS_{CFG}$	**15.8**	**15.0**	**17.2**	**16.6**	**16.6**	**15.4**
	$FISS_{CB}$	**6.6**	**7.6**	**8.2**	6.6	6.8	**8.8**
	$FISS_{IC}$	6.4	**7.8**	**6.6**	7.2	5.4	5.6
	$FISS_{PB}$	6.2	**7.0**	**6.6**	6.2	5.6	5.4
	$FISS_{CT}$	6.4	**7.4**	**8.8**	6.6	8.4	**9.6**

The top 3 performing methods are shaded.
Darker shades indicating better performance.

default parameters without any fine-tuning. Extraction of code representation and corpus collection is implemented in Python. To mitigate the effect of randomness, all experimental results related to inconsistencies were averaged over five repetitions of experiments, with each experiment set to run for 24 hours. Experiments were conducted on a server with two dodeca-core CPUs Intel(R) Xe on(R) Silver 4214 CPU @ 2.20GHz and 251GB RAM, running Ubuntu 18.04.4 LTS (64-bit).

V. RESULT ANALYSIS

A. RQ1: Performance of Initial seed Selection

Effectiveness of initial seed selection methods. The budget of existing corpus minimization methods (i.e., PeachSet, MinSet, and OptiMin) applied to P1 and P2 is mainly between 20% and 50%. Inspired by this, to reduce the fuzzing budgets, we initially selected subsets with budgets set at 20%, 35%, and 50%. Then, we applied each studied initial seed selection method with three budgets on benchmark corpus P1 and P2. JavaTailor and VECT conducted 24 hours of fuzzing on each subset, counting the number of detected unique inconsistencies. Table IV shows the comparison results of each method in terms of unique inconsistencies in the differential testing. The FullSet and RandomSet in the second column represent the use of the entire set of initial seeds and the subset selected by random selection, serving as baselines for comparison. Results highlighted in bold indicate superior performance compared to the FullSet. The top three performing methods are shaded, with darker shades indicating better performance.

Results. From Table IV, we can observe that (1) When compared to the full set (FullSet), all three types of initial seed selection methods can outperform the full set within the given fuzzing time. This finding confirms the necessity of performing initial seed selection in JVM fuzzing. (2) When compared to random selection (RandomSet), evaluation results suggest that randomly selecting a subset could have a negative impact, which underlines the importance of carefully designing selection strategies. (3) When compared across the studied selection methods, $FISS_{CFG}$ is the best-performing method. It outperforms FullSet in all results, yielding between 1.42 and 2.69 times higher performance, and ranks in the top three in all column comparisons. For example, in JavaTailor and P2, the number of unique inconsistencies for the three budgets is 15.8, 15.0, and 17.2 respectively, compared to 6.4 when using FullSet. The possible reason is that semantic information effectively represents the feature of the program, thereby enhancing the effectiveness of sorting in FPS. $FISS_{AST}$ follows behind $FISS_{CFG}$, as it extracts features using AST, which is likely to have a coarser granularity compared to CFG. The pre-trained model methods, such as $FISS_{CB}$ and $FISS_{IC}$, tend to exhibit relatively unstable and inferior performance. This could potentially be attributed to these models not being fine-tuned specifically for initial seed selection tasks. Meanwhile, we noted that while coverage-based methods excel in traditional program fuzzing, the extensive code space within the JVM poses a challenge. $CISS_P$ and $CISS_M$ are likely unable to effectively leverage coverage for assessing the quality of seed programs, suggesting that a significant portion of the coverage may not be relevant for inconsistency detection.

> **Finding I**: The initial seed selection that utilizes control flow graph, $FISS_{CFG}$, outperforms all other studied methods, yielding between 1.42 and 2.69 times the results of FullSet within the same test time, with an average improvement of 101%.

Effectiveness of budget selection. It is unknown whether the established budget sizes from the existing corpus minimization are the most effective for JVM fuzzing. Particularly, we study a variety of initial seed selection methods and certain strategies may perform better given a specific budget. Hence, we extend our investigation to include a wider budget range, spanning from 5% to 95%. For our exploratory analysis, we specifically evaluated the effectiveness of these budgets on the P1 corpus and the VECT fuzzer. Table V shows the average results of five repeated experiments, with shadows representing the best performers in each row.

Results. From Table V, we can observe that (1) Regardless of the budget size, the studied initial seed selection methods

Table V: The number of inconsistencies with different budgets

Method	5%	20%	35%	50%	65%	80%	95%
CISS_M	7.7	7.2	9.4	8.4	6.3	8.0	7.0
CISS_P	4.0	5.4	6.6	7.4	8.7	7.7	8.0
PISS	7.2	9.4	8.6	7.8	9.2	7.0	5.8
FISS_{TS}	5.0	6.6	7.6	7.6	7.2	7.4	7.0
FISS_{AST}	3.2	8.6	8.6	10	8.2	6.4	8.0
FISS_{CFG}	6.0	12.8	14.8	16.2	11.2	9.4	8.6
FISS_{CB}	5.2	6.2	7.4	8.2	7.6	9.4	7.8
FISS_{IC}	5.6	7.8	7.8	8.4	9.4	8.6	6.8
FISS_{PB}	5.0	7.6	7.8	7.2	9.6	7.6	8.2
FISS_{CT}	7.4	7.2	8.6	8.2	8.8	8.6	8.2

Table VI: Time overhead of each method in P1

Method	Data Collection	Data Processing	Seed Selection	Sum
CISS_M	188,330 s	36,779 s	819 s	225,928 s
CISS_P	188,330 s	36,779 s	41 s	225,150 s
PISS	140,700 s	1 s	1 s	140,702 s
FISS_{TS}	0 s	20 s	6 s	26 s
FISS_{AST}	0 s	3 s	1 s	4 s
FISS_{CFG}	0 s	29 s	1 s	30 s
FISS_{CB}	0 s	395 s	1 s	396 s
FISS_{IC}	0 s	272 s	1 s	273 s
FISS_{PB}	0 s	477 s	1 s	478 s
FISS_{CT}	0 s	878 s	1 s	879 s

relatively achieve superior performance when compared to the full set and the random selection. (2) Most coverage-based, HotSet-based, and traditional feature-based methods tend to perform better with smaller budgets. For instance, the number of inconsistencies for CISS_M and FISS_{CFG} ranges from 7.2 to 9.4 and 12.8 to 16.2 respectively when the budgets are between 20% and 50%, while the number of inconsistencies ranges from 6.3 to 8.0 and 8.6 to 11.2 respectively when the budgets are between 65% and 95%. Conversely, the four pre-trained model methods are likely to achieve better performance with relatively large budgets. For example, FISS_{IC}, FISS_{PB}, and FISS_{CT} achieve the best performance when the budget is 65%, with the number of inconsistencies being 9.4, 9,6, and 8,8, respectively. The potential reason is that these models were not specifically optimized for tasks related to selecting initial seeds. (3) FISS_{CFG} consistently delivers the best performance across most (except 5%) budget ranges, demonstrating its effectiveness in JVM fuzzing. When the budget for FISS_{CFG} is set between 20% and 50%, the top three results can be achieved, further confirming the initial budget setting choice in prior experiment. We suggest setting the FISS_{CFG} budget to 50%, as it provides the best outcomes across various budgets.

> **Finding II**: FISS_{CFG} continuously achieves the top performance across a wider range of budget settings. Specifically, We recommend using FISS_{CFG} with a budget set of 50% for initial seed selection, which has been found to achieve optimal performance.

Efficiency of initial seed selection methods. Table VI presents the time overhead of each initial seed selection method on corpus P1, including three phases: data collection, data processing, and initial seed selection. As similar trends are observed across different corpus, we only provided the results on P1. Note that the budget setting and fuzzing techniques do not affect the time overhead, as we sort the seed program first.

Results. As shown in the table, the total time overhead of all the traditional feature-based methods requires no more than 30 seconds to select corpus subsets (i.e., 26s, 4s, and 30s for FISS_{TS}, FISS_{AST}, and FISS_{CFG}, respectively) which

is significantly faster than the other methods. For the methods based on pre-trained models, results show that most of the time is spent on data processing when compared to traditional ones. For the prefuzz-based method PISS, a considerable amount of time (i.e., 140,700s) is taken to collect data as it necessitates fuzzing each seed program for five minutes. Coverage-based methods CISS exhibit the highest time overhead across all three phases, taking over 60 hours in total as it collects the coverage of each seed program for more than 500 seconds.

> **Finding III**: Traditional feature-based methods have a significantly smaller time overhead compared to other methods. More specifically, FISS_{CFG} not only enhances the effectiveness of JVM fuzzing but also demonstrates its efficiency.

B. RQ2: Initial Seed Selection on Programs in the Wild

To evaluate the effectiveness of initial seed selection on programs in the wild, we first confirm the importance of the collected open-source corpus (P3) by analyzing the overlap of unique inconsistencies detected across the studied corpus (i.e., P1, P2, and P3). Then, we validate the generalizability of initial seed selection by repeating the experiments of RQ1 on the P3 corpus. For the first analysis, we conducted fuzzing on FullSet by employing the same experimental settings from RQ1. For each corpus, we collated all inconsistencies detected by two fuzzers over five repeated runs, and then eliminated duplicates based on crash messages. For the second analysis, we applied each studied initial seed selection method on P3 with a wider budget range, i.e., spanning from 5% to 95%. We also performed an overlap analysis for the best-performing method. The Venn diagrams in Figure 2 show the results of the inconsistency overlap analysis, while Table VII records the results of the performance at various budgets.

Results. From Figure 2a, we found that each studied corpus can detect some unique inconsistencies with the full set, i.e., 19, 14, 8 for P1, P2, and P3, respectively. This finding suggests that the open-source corpus and the existing benchmark corpus complement each other. In addition, the results show that P3 detected a total of 17 unique inconsistencies, while P1 and
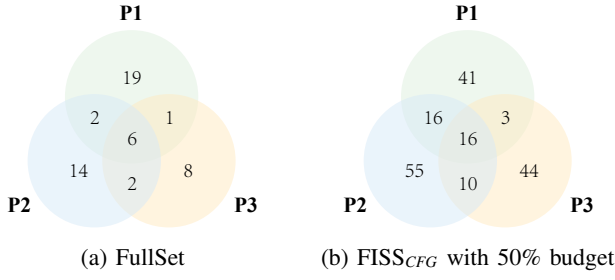
(a) FullSet   (b) FISS$_{CFG}$ with 50% budget

Figure 2: Inconsistency overlap

Figure 3: Trend of bugs detected by FISS$_{CFG}$ and FullSet

P2 were able to detect 28 and 24 inconsistencies separately. This indicates that the open-source corpus is less effective in detecting unique inconsistencies and the quality of its seed program is relatively lower.

From Table VII, we observed that on the open-source corpus, FISS$_{CFG}$ still emerges as the best selection method, consistently ranking in the top three, with an improvement of 8.3% to 108% compared to the full set. For example, when the budget is set at 50%, the number of detected unique inconsistencies with FISS$_{CFG}$ reaches 10.4 for JavaTailor and 9.6 for VECT. Besides, selection methods relying on pre-trained models show substantial performance enhancements. A possible reason is that, similar to P3, pre-trained models also use GitHub as a data source, which could potentially enhance the representation of code semantics. This further illustrates that the lack of fine-tuning leads to the poor performance of pre-trained model-based methods on P1 and P2.

From Figure 2b, we noticed that the most effective seed selection method (FISS$_{CFG}$ with a 50% budget) allows P3 to identify a comparable number of unique inconsistencies as P1 and P2. For instance, the number of unique inconsistencies detected with P1, P2, and P3 is 41, 55, and 44, respectively. Moreover, the number of unique inconsistencies detected by P3 improves significantly when compared to the full set, i.e., 73 and 17 for FISS$_{CFG}$ and FullSet, respectively. The above results suggest that the initial seed selection can enhance the effectiveness of programs in the wild and further improve the fuzzing capability.

> **Finding IV**: Programs in the wild complement widely-studied initial seeds by detecting new JVM behaviors. Selection methods relying on pre-trained models achieve substantial performance improvements in the open-source corpus. Nonetheless, FISS$_{CFG}$ remains the best-performing method, with an improvement of 8.3% to 108%, and enhances the effectiveness of the open-source corpus.

C. RQ3: Detection of Previously Unknown Bugs

We further investigated whether the initial seed selection helps enhance JVM fuzzing techniques in detecting previously unknown bugs through the differential-testing experiments on the latest builds. Specifically, we applied VECT on all studied corpus, as it has been proven to be state-of-the-art [3]. For a
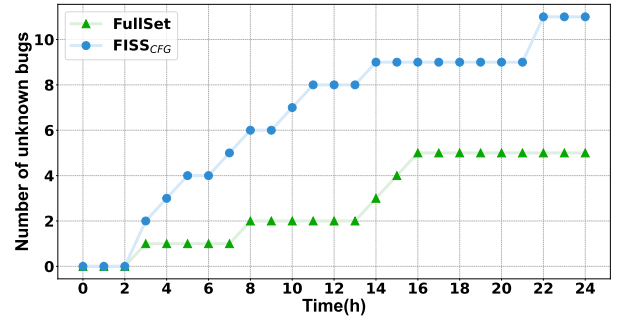
fair comparison, we ran VECT 24 hours on both the FullSet and the subsets selected by each method with a 50% budget. We then submitted the detected inconsistencies on the latest builds to the corresponding developers to check whether it was a real bug. Table IX shows the total number of bugs discovered by each initial seed subset across the three studied corpus within the same testing period, and Figure 3 shows the trend of bugs for FullSet and FISS$_{CFG}$.

Results. As shown in Talbe IX, FISS$_{CFG}$ is the best-performing method across all the studied selection methods, yielding between 1.4 and 2.2 times higher performance. Figure 3 further shows the trend of the number of bugs detected by FullSet and FISS$_{CFG}$ over time, where the x-axis represents the testing time, while the y-axis represents the number of unknown bugs detected by FullSet and FISS$_{CFG}$ within the corresponding testing time. As shown in Figure 3, FISS$_{CFG}$ detected more unknown bugs than FullSet during the entire testing process, specifically 11 and 5 bugs respectively. In the first 12 hours of fuzzing, FISS$_{CFG}$ was able to detect unknown bugs more quickly. This is anticipated as FISS$_{CFG}$ selects a subset with higher quality for the corpus, resulting in enhanced bug discovery capabilities. Notably, all five bugs detected by FullSet were also detected by FISS$_{CFG}$, further emphasizing the superiority of initial seed selection.

We also conducted a large-scale fuzzing experiment by applying FullSet and FISS$_{CFG}$ for each corpus to VECT with a total of over 2,000 hours, which aimed to evaluate the time overhead required by FullSet to detect the same number of unknown bugs as FISS$_{CFG}$. We found that FullSet detecting all the bugs detected by FISS$_{CFG}$ required over 1,000 hours in total. This further demonstrates that the initial seed selection method FISS$_{CFG}$ significantly improves JVM fuzzing performance.

In total, 25 previously unknown bugs were detected by our initial seed selection methods in our experiments, 21 of which have been confirmed and fixed by developers. Table VIII shows the information about these 21 bugs. Among these, 7 were bugs that could only be detected by P3, further highlighting the significance of the open-source corpus in enhancing JVM fuzzing techniques. We then used a previously unknown bug detected by P3 as an example to illustrate the effectiveness of open-source corpus. Figure 4 shows a simplified synthesized test program that triggers a runtime check bug [50] in OpenJ9,

Table VII: Comparison results on programs in the wild in terms of the number of unique inconsistencies

		JavaTailor							VECT						
	FullSet	5.0							4.8						
		5%	20%	35%	50%	65%	80%	95%	5%	20%	35%	50%	65%	80%	95%
	RandomSet	4.3	5.2	4.4	5.6	5.0	4.8	5.3	4.2	4.0	3.8	4.2	5.3	5.0	5.4
	$CISS_M$	5.0	5.0	5.2	4.8	6.7	5.3	5.0	5.0	3.8	4.8	4.4	6.3	5.8	5.3
	$CISS_P$	3.3	4.2	4.6	4.4	7.3	4.7	5.3	3.0	3.6	5.2	3.8	5.8	6.3	4.5
P3	PISS	7.0	7.6	5.8	5.4	7.0	5.7	4.8	4.7	6.8	7.0	7.6	9.0	6.7	5.0
	$FISS_{TS}$	4.3	6.8	5.6	6.8	9.0	7.0	5.5	3.3	4.8	5.2	6.8	7.3	6.5	5.5
	$FISS_{AST}$	3.3	6.0	7.8	7.0	8.3	6.3	6.0	3.0	6.0	6.6	7.0	5.5	6.8	5.8
	$FISS_{CFG}$	8.0	8.8	7.4	10.4	10.0	8.4	5.6	5.2	7.4	6.6	9.6	9.6	7.4	5.5
	$FISS_{CB}$	7.5	7.8	7.0	7.6	8.5	7.0	5.0	4.7	6.2	6.0	6.2	7.8	7.3	5.3
	$FISS_{IC}$	8.6	10.2	8.8	7.4	8.5	7.8	5.8	6.8	9.2	8.0	4.2	7.5	7.3	5.0
	$FISS_{PB}$	4.3	6.2	7.4	4.8	8.0	6.7	5.3	2.5	3.6	3.8	2.4	7.3	6.5	5.0
	$FISS_{CT}$	8.8	5.4	6.4	7.8	8.8	7.3	4.8	5.8	6.2	5.6	6.4	6.5	5.5	4.8

Table VIII: Confirmed or fixed unknown bugs

Bug ID	JVM	Affected OpenJDK	Status	Corpus
#15061	OpenJ9	11	fixed	P1
#17247	OpenJ9	17	fixed	P1
#17248	OpenJ9	17	fixed	P1,P2,P3
#19014	OpenJ9	11,17	fixed	P1
#19015	OpenJ9	8,11,17	fixed	P2
#19016	OpenJ9	8,11,17	fixed	P3
#19124	OpenJ9	8,11,17	confirmed	P1
#19125	OpenJ9	8,11,17	confirmed	P1,P2,P3
#19129	OpenJ9	8	fixed	P1,P2
#19130	OpenJ9	8,11,17	confirmed	P1
#19132	OpenJ9	17	fixed	P3
#19139	OpenJ9	8	confirmed	P1
#19140	OpenJ9	8,11,17	fixed	P1
#19163	OpenJ9	17	confirmed	P1
JDK-8326996	Hotspot	11,17	confirmed	P3
JDK-8327011	Hotspot	8	confirmed	P3
JDK-8327012	Hotspot	17	confirmed	P1,P2
JDK-8328298	Hotspot	11,17	confirmed	P3
#I98GAP	Bisheng JDK	17	confirmed	P1,P2
#I98GCO	Bisheng JDK	8	confirmed	P3
#I98GD8	Bisheng JDK	11	confirmed	P3

```
1  public static bool failed = false;
2  public static void main(String[] var0){
3      try{
4          ...
5          int var0 = null.test();
6          ...
7      }catch (Throwable e){
8          failed = true;
9      }
10     if (failed){
11         Class var1 = new TestCase$1().getClass();
12         Class var2 = var1.getEnclosingClass();
13         ...
14     }
15 }
```

Figure 4: OpenJ9 bug #19016

of OpenJ9 have confirmed and fixed this bug. Note that this bug only can be detected by P3, since this bug requires the seed program to new an anonymous inner class and call the `getEnclosingClass` function.

> **Finding V**: Initial seed selection enhances JVM techniques for detecting more previously unknown bugs given the same testing period, with 21 being already confirmed and fixed by developers. Seven of them can only be found by the open-source corpus.

VI. IMPLICATIONS AND FUTURE WORK

Designing a task-specific initial seed selection method is crucial. As shown in RQ1 and RQ2, methods widely evaluated in traditional software testing are not suitable for JVM testing due to the characteristics of large-scale and intricate code in JVM. Inspired by existing work, we propose program-feature-based selection methods and demonstrate their superiority. These findings suggest a need to design task-specific seed selection methods, as different test subjects often exhibit unique characteristics. For efficiency and effectiveness, we recommend using the $FISS_{CFG}$ method, which demonstrated the most practical in our JVM testing experiments.

as detected by P3. In this example, lines shaded green (Lines 4,5) represent the ingredient that was inserted, while lines shaded blue (Lines 11,12) represent the root cause of the bug. The original seed program code does not throw any exception, so the `failed`'s value is false, which causes the branch containing the bug not to be executed. VECT inserts an ingredient containing a `NullPointerException` which causes the value of `failed` to be changed to true (Line 8). In the branch, the synthetical program first news an anonymous inner class (i.e., `TestCase$1`) and tries to get its enclosing class (Line 12). If `TestCase$1` is a normal inner class, OpenJ9 needs to check whether the inner class and its enclosing class have the same `InnerClass` attribute. However, OpenJ9 incorrectly applies this check to anonymous inner classes and throws an `IncompatibleClassChangeError`. The developers

Table IX: Comparison results of different methods in terms of the number of previously unknown bugs

Method	FullSet	CISS$_M$	CISS$_P$	PISS	FISS$_{TS}$	FISS$_{AST}$	FISS$_{CFG}$	FISS$_{CB}$	FISS$_{IC}$	FISS$_{PB}$	FISS$_{CT}$
#Unknown Bugs	5	6	6	5	6	8	11	8	7	5	7

Selecting an optimal corpus budget is significant. The results shown in Table V indicate that different initial seed selection methods tend to achieve the best performance at varying corpus budgets. Specifically, we recommend using FISS$_{CFG}$ with a 50% budget for initial seed selection, which has been found to optimize performance and resource utilization in JVM fuzzing. Future work could focus on refining and implementing an adaptive budget selection algorithm for FISS$_{CFG}$, in order to determine the minimum corpus budget required for the best performance.

Exploring open-source corpus for improving JVM fuzzing is beneficial. Existing work often used the same corpus as older works for fuzzing, even when proposing new techniques, thus neglecting the significance of the diverse corpus. Our RQ2 results confirmed that different corpus exhibit complementary effectiveness by detecting new behaviors and bugs. Significantly, the findings reveal that the initial seed selection can improve the quality of the open-source corpus. Hence, future work could further mix all corpus for further selection and integrate their diverse testing capability to enhance overall effectiveness.

Fine-tuning the pre-trained model for initial seed selection shows promise. As shown in table IV, pre-trained-model-based FISS achieves substantial performance enhancements on the open-source corpus compared to the benchmark corpus. This is because the open-source corpus data is collected from GitHub, which enhances the learning of code semantics. To further improve effectiveness, future work will involve fine-tuning the pre-trained code representation models across various programming languages and compilers to better fit the downstream task of initial seed selection.

VII. THREATS TO VALIDITY

External threats to validity primarily lie in the corpus and fuzzers used in our study. Firstly, we eliminated seed programs that can directly identify inconsistencies to prevent interference with fuzzing results. In the future, we plan to incorporate additional corpus to further mitigate the threat. Secondly, although these initial seed selection methods can be generalized to any JVM fuzzer that takes seeds as input, we only evaluated the impact of initial seed selection on JavaTailor and VECT. We selected them as the representatives due to their state-of-the-art effectiveness [3], [51] and general testing purposes (e.g., JITFuzz [17] targets the JIT component only).

Construct threats to validity mainly stem from the duration of each fuzzing process and its randomness. Despite our best efforts to assess the impact of initial seed selection and open-source corpus on JVM fuzzing effectiveness, the fuzzing time we have set is considerably shorter than that in industrial fuzzing. Due to resource constraints, we are unable to extend the fuzzing time for each iteration. Besides, the execution of the fuzzer and breaking ties involve randomness. To reduce the impact of this randomness, we performed each experiment five times using different random seeds.

Internal threat to validity mostly lies in the implementations of each technique. To mitigate this threat, we relied on the available APIs to extract AST and CFG and utilized the pre-trained code representation models with their default settings.

VIII. CONCLUSION

This work designed a total of 10 initial seed selection methods (i.e., coverage-based, prefuzz-based, and program-feature-based), in order to enhance the JVM fuzzing effectiveness. We conducted an empirical study on three JVM implementations with JavaTailor and VECT to comprehensively evaluate the performance of initial seed selection methods. The results highlight that the subset selected by initial seed selection outperforms the entire set of initial seeds. In particular, the method utilizing control flow graphs performs the best. Furthermore, the study emphasizes the benefits of incorporating programs in the wild, demonstrating the complementary effectiveness with the existing benchmark corpus. In addition, given the same testing period, initial seed selection can enhance fuzzing techniques by detecting more previously unknown bugs, with 21 bugs having been confirmed or fixed by developers. Our work also opens up several promising future directions including determining the minimum corpus budget, fine-tuning pre-trained code representation models for better fitting the downstream task, and mixing diverse corpus for selection.

Data Availability. The replication package that supports the findings of this study is available publicly [12].

REFERENCES

[1] Y. Chen, T. Su, and Z. Su, "Deep differential testing of jvm implementations," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1257–1268.

[2] Y. Zhao, Z. Wang, J. Chen, M. Liu, M. Wu, Y. Zhang, and L. Zhang, "History-driven test program synthesis for jvm testing," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1133–1144.

[3] T. Gao, J. Chen, Y. Zhao, Y. Zhang, and L. Zhang, "Vectorizing program ingredients for better jvm testing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 526–537.

[4] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 861–875.

[5] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, "Seed selection for successful fuzzing," in *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*, 2021, pp. 230–243.

[6] Eddington, "Peach fuzzer," 2024, http://peachfuzzer.com/.

[7] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[8] H. Jia, M. Wen, Z. Xie, X. Guo, R. Wu, M. Sun, K. Chen, and H. Jin, "Detecting jvm jit compiler bugs via exploring two-dimensional input spaces," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 43–55.

[9] "Hotspot," 2024, http://openjdk.java.net.

[10] "Openj9," 2024, https://www.eclipse.org/openj9

[11] "Bisheng," 2024, https://www.openeuler.org/zh/other/projects/bishengjdk.

[12] "Selecting initial seeds for better jvm fuzzing," 2024, https://github.com/gaotravor/Initial-seed-selection.

[13] H. Wang, J. Chen, C. Xie, S. Liu, Z. Wang, Q. Shen, and Y. Zhao, "Mlirsmith: Random program generation for fuzzing mlir compiler infrastructure," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1555–1566.

[14] H. You, Z. Wang, J. Chen, S. Liu, and S. Li, "Regression fuzzing for deep learning systems," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 82–94.

[15] H. Ma, Q. Shen, Y. Tian, J. Chen, and S.-C. Cheung, "Fuzzing deep learning compilers with hirgen," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 248–260.

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

[17] M. Wu, M. Lu, H. Cui, J. Chen, Y. Zhang, and L. Zhang, "Jitfuzz: Coverage-guided fuzzing for jvm just-in-time compilers," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 56–68.

[18] M. Wu, Y. Ouyang, M. Lu, J. Chen, Y. Zhao, H. Cui, G. Yang, and Y. Zhang, "Sjfuzz: Seed and mutator scheduling for jvm fuzzing," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1062–1074.

[19] OpenJDK, "Javafuzzer," 2024, https://github.com/shipilev/JavaFuzzer.

[20] S. Hwang, S. Lee, J. Kim, and S. Ryu, "Justgen: Effective test generation for unspecified jni behaviors on jvms," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1708–1718.

[21] Z. Zang, N. Wiatrek, M. Gligoric, and A. Shi, "Compiler testing using template java programs," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.

[22] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2123–2138.

[23] Mozilla, "Fuzzing-test samples," 2024, https://firefox-source-docs.mozilla.org/tools/fuzzing/index.html.

[24] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse, "Adaptive random test case prioritization," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 233–244.

[25] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing. in 2017 ieee/acm 39th international conference on software engineering (icse)," *IEEE, 700–711*, 2017.

[26] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Test case prioritization for compilers: A text-vector based approach," in *2016 IEEE international conference on software testing, verification and validation (ICST)*. IEEE, 2016, pp. 266–277.

[27] E. Cruciani, B. Miranda, R. Verdecchia, and A. Bertolino, "Scalable approaches for test suite reduction," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 419–429.

[28] R. Pan, T. A. Ghaleb, and L. Briand, "Atm: Black-box test case minimization based on test code similarity and evolutionary search," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1700–1711.

[29] H. Green and T. Avgerinos, "Graphfuzz: Library api fuzzing with lifetime-aware dataflow graphs," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1070–1081.

[30] K. Sparck Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of documentation*, vol. 28, no. 1, pp. 11–21, 1972.

[31] E. Foundation, "Eclipse jdt (java development tools)," 2024, https://projects.eclipse.org/projects/eclipse.jdt.

[32] A. R. Golding and Y. Schabes, "Combining trigram-based and feature-based methods for context-sensitive spelling correction," *arXiv preprint cmp-lg/9605037*, 1996.

[33] S. R. Group, "Soot: A framework for analyzing and transforming java and android applications," 2024, https://sable.github.io/soot.

[34] L. Yang, J. Chen, H. You, J. Han, J. Jiang, Z. Sun, X. Lin, F. Liang, and Y. Kang, "Can code representation boost ir-based test case prioritization?" in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2023, pp. 240–251.

[35] M. Yan, J. Chen, J. M. Zhang, X. Cao, C. Yang, and M. Harman, "Coco: Testing code generation systems via concretized instructions," *arXiv preprint arXiv:2308.13319*, 2023.

[36] C. Yang, J. Chen, B. Lin, J. Zhou, and Z. Wang, "Enhancing llm-based test generation for hard-to-cover branches via program analysis," *arXiv preprint arXiv:2404.04966*, 2024.

[37] Z. Tian, J. Chen, and X. Zhang, "On-the-fly improving performance of deep code models via input denoising," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 560–572.

[38] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[39] N. D. Bui, Y. Yu, and L. Jiang, "Infercode: Self-supervised learning of code representations by predicting subtrees," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1186–1197.

[40] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[41] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[42] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making: 9th Asian Computing Science Conference. Dedicated to Jean-Louis Lassez on the Occasion of His 5th Birthday. Chiang Mai, Thailand, December 8-10, 2004. Proceedings 9*. Springer, 2005, pp. 320–329.

[43] R. Huang, W. Sun, Y. Xu, H. Chen, D. Towey, and X. Xia, "A survey on adaptive random testing," *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2052–2083, 2019.

[44] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse, "Adaptive random testing: The art of test case diversity," *Journal of Systems and Software*, vol. 83, no. 1, pp. 60–66, 2010.

[45] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "Pointnet++: Deep hierarchical feature learning on point sets in a metric space," *Advances in neural information processing systems*, vol. 30, 2017.

[46] C. Yang, J. Chen, X. Fan, J. Jiang, and J. Sun, "Silent compiler bug de-duplication via three-dimensional analysis," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 677–689.

[47] Y. Li, R. Bu, M. Sun, W. Wu, X. Di, and B. Chen, "Pointcnn: Convolution on x-transformed points," *Advances in neural information processing systems*, vol. 31, 2018.

[48] "Requests: Http for humans," 2024, https://requests.readthedocs.io/en/latest/.

[49] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 180–190.

[50] "Openj9 issue #19016," 2024, https://github.com/eclipse-openj9/openj9/issues/19016.

[51] Z. Zang, F.-Y. Yu, A. Thimmaiah, A. Shi, and M. Gligoric, "Java jit testing with template extraction," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1129–1151, 2024.