Detecting JVM JIT Compiler Bugs via Exploring Two-Dimensional Input Spaces

Haoxiang Jia[1†], Ming Wen[1†*], Zifan Xie[1†], Xiaochen Guo[1†] Rongxin Wu[2]
Maolin Sun[1†], Kang Chen[1†], Hai Jin[3†]

[1]*School of Cyber Science and Engineering, Huazhong University of Science and Technology, China*
[2]*School of Informatics, Xiamen University, China*
[3]*School of Computer Science and Technology, Huazhong University of Science and Technology, China*
Email: {haoxiangjia, mwenaa, xzff, xiaochenguo, merlinsun, void0red, hjin}@hust.edu.cn, {wurongxin}@xmu.edu.cn

Abstract—**Java Virtual Machine (JVM) is the fundamental software system that supports the interpretation and execution of Java bytecode. To support the surging performance demands for the increasingly complex and large-scale Java programs, Just-In-Time (JIT) compiler was proposed to perform sophisticated runtime optimization. However, this inevitably induces various bugs, which are becoming more pervasive over the decades and can often cause significant consequences. To facilitate the design of effective and efficient testing techniques to detect JIT compiler bugs. This study first performs a preliminary study aiming to understand the characteristics of JIT compiler bugs and the corresponding triggering test cases. Inspired by the empirical findings, we propose `JOpFuzzer`, a new JVM testing approach with a specific focus on JIT compiler bugs. The main novelty of `JOpFuzzer` is embodied in three aspects. First, besides generating new seeds, `JOpFuzzer` also searches for diverse configurations along the new dimension of optimization options. Second, `JOpFuzzer` learns the correlations between various code features and different optimization options to guide the process of seed mutation and option exploration. Third, it leverages the profile data, which can reveal the program execution information, to guide the fuzzing process. Such novelties enable `JOpFuzzer` to effectively and efficiently explore the two-dimensional input spaces. Extensive evaluation shows that `JOpFuzzer` outperforms the state-of-the-art approaches in terms of the achieved code coverages. More importantly, it has detected 41 bugs in OpenJDK, and 25 of them have already been confirmed or fixed by the corresponding developers.**

Index Terms—**JVM, JIT Compiler, JVM Testing**

I. INTRODUCTION

Java Virtual Machine (JVM) plays a fundamental role in supporting the interpretation and execution of Java bytecode, which can be compiled from various high-level programming languages such as Scala, Java, and Kotlin. Due to the extensive utilization of the Java programming language [1] and the active development of bytecode-based applications, many JVM systems have been implemented by different organizations and companies, such as the HotSpot by Oracle [2], DragonWell by Alibaba [3], OpenJ9 by IBM [4], and Zulu by Azul [5]. Such JVM systems are often complex in their functionalities and large-scale in sizes [6], and thus critical bugs or vulnerabilities

are inevitable, which often lead to unexpected behaviors or even catastrophic consequences for end users. Therefore, ensuring the quality of JVM systems is critical.

As the size of the programs running on top of JVM becomes increasingly larger, greater demands are being placed on the performance of JVM systems. Catering to the surging demands for the performance of Java applications, Just-In-Time (JIT) compiler was developed as a key component of JVM (e.g., OpenJDK supports JIT since Java Version 1.3), and performs *optimizations* to generate high quality machine code during runtime. Typically, there are two main JIT compilers, namely, the client compiler (*a.k.a* the **C1** compiler) and the server compiler (*a.k.a* the **opto** or **C2** compiler). Common optimization strategies include *method inlining*, *escape analysis*, and *loop unrolling*. Via inspecting all the existing bug reports of HotSpot, a popular implementation of JVM, we observe that the total number of reported bugs has decreased over the last ten years, while the proportion of JIT compiler-related ones has increased greatly. Such bugs can have significant impacts, such as leading JVM systems to crash [7].

Various fuzzing techniques have been proposed to detect JVM bugs recently [8]–[10]. The basic intuition is to generate diverse and effective test inputs, e.g., source files (`*.java`) or bytecode files (`*.class`), to test JVM systems to trigger unexpected behaviors. For instance, *classfuzz* employs a set of pre-defined syntactic mutation operators, such as deleting exception handlers and changing variable modifiers/types [8], to generate test inputs. *classming* was proposed to manipulate the control/data flow via inserting/deleting `goto` or `return` statements to generate diverse test inputs [9]. More recently, *JavaTailor* was proposed [10] to synthesize seeds via extracting ingredients from historical bug-triggering test cases. Despite the promising results, their effectiveness in detecting JIT compiler bugs is, unfortunately, significantly compromised for the following reasons. First, they can only detect limited types of JVM bugs that are irrelevant to optimizations. For instance, the test inputs generated by *classfuzz* can only test the *loading*, *linking*, and *initialization* phases in JVMs, while deeper logic, such as code optimization, can rarely be explored [8]. Second, they mainly focus on mutating class files while we observe that substantial JIT compiler bugs can only be triggered under specific optimization configurations (see Section III-B).

Therefore, merely focusing on the dimension of mutating class files is insufficient for triggering such optimization bugs.

To devise tools that can detect JIT compiler bugs more effectively, we first perform a preliminary study. Specifically, we extracted 9,252 previously reported JIT compiler bugs of HotSpot from the JDK bug tracking system and made the following findings. First, 61.7% of the JIT compiler bugs can only be triggered under certain optimization options with non-default values. In addition, the input class files that can trigger existing JIT bugs often contain a larger number of specific code features such as `assign` statements, `field` access, `loop` statements and `arithmetic` operators. Furthermore, we also observe that the profile data, which records the information of dynamic program execution, can reflect to what extent an input class file has been optimized.

Inspired by our empirical findings, we design `JOpFuzzer`, a new JVM testing approach to detecting JIT compiler bugs. The major novelties of `JOpFuzzer` are embodied in three aspects. First, besides generating new seeds as adopted by existing approaches [8]–[11], `JOpFuzzer` also searches for diverse configurations along the new dimension of optimization options. However, exploring seed and optimization options collectively will significantly enlarge the search space, and thus might compromise both the effectiveness and efficiency of existing fuzzing techniques. As revealed by our preliminary study, a certain optimization bug can only be triggered when using specific code features together with certain options under specific values. Therefore, considering the possible values of various options as well as the diversity of code features, the search space will be explosive. The other two novelties of `JOpFuzzer` aim at addressing this challenge. Specifically, `JOpFuzzer` first learns the correlations between various code features and different optimization options based on a large set of regression tests in prior to the fuzzing process. Such prior knowledge constructed can guide the process of seed mutation and option selection during the fuzzing process. Finally, it also leverages the profile data, which can reveal the program execution information, to guide the process of *seed scheduling*.

We performed comprehensive experiments to evaluate the effectiveness and usefulness of `JOpFuzzer`. Our results show that `JOpFuzzer` can outperform the state-of-the-art approaches in terms of code coverage. In particular, with respect to line coverage, `JOpFuzzer` outperforms the state-of-the-art JVM testing tools, *JavaTailor* and *classming*, by 20.3% and 33.5% respectively on OpenJDK11. On OpenJDK17, the corresponding improvements are 21.6% and 30.4% respectively. Besides, the improvements over function coverage are also significant. More importantly, `JOpFuzzer` has detected 41 bugs in OpenJDK, 25 of which have already been confirmed or fixed by the corresponding developers.

This study makes the following major contributions:

- **Empirical Study:** We performed an empirical investigation to understand the characteristics of JVM JIT compiler bugs and their triggering test cases. The study reveals important findings on the detection of JIT compiler bugs and can shed lights on future researches.

- **Approach:** Inspired by our empirical findings, we designed `JOpFuzzer`, which can detect JIT Compiler bugs effectively and efficiently via exploring two-dimensional input spaces. Experimental evaluation demonstrates that it can outperform existing approaches.
- **Usefulness:** `JOpFuzzer` has detected many real issues on OpenJDK. In particular, we reported 41 bugs, with 25 of them already confirmed or fixed by developers. Such results reflect the usefulness of our approach.
- **Artifact:** We open-sourced the data of our study as well as the approach to facilitate future related researches. The artifact and bug reports are available at: **https://github.com/CGCL-codes/JOpFuzzer**.

II. BACKGROUND & MOTIVATION

A. JVM and JIT Compiler

Typically, the JVM performs interpreter execution (i.e., interpreting bytecode files (`*.class`) at the initial stage). Meanwhile, it will also spot methods or basic blocks that are frequently executed, and mark these parts as the *HotSpot* code [12]. In order to improve the efficiency of executing the HotSpot code, JVM compiles it into high-quality machine code during runtime and optimizes it with sophisticated strategies. The compiler that implements this process is called the Just-In-Time (JIT) compiler [13], which is an important component of many JVM implementations (e.g., the HotSpot of OpenJDK). Currently, there are two mainstream JIT compilers, namely, the *C1 Compiler* (client compiler) and *C2 Compiler* (server compiler). The former enjoys a higher startup speed but with lower peak performance and mainly comprises local optimizations such as inlining. On the contrary, the *C2 Compiler* focuses on global optimizations as well as certain unreliable or aggressive optimizations based on the program runtime information (stack state and branch execution information), resulting in longer startup time for long-running background programs.

B. JIT Compiler Bugs

The JIT compiler is much more sophisticated than the *javac* compiler [14], and thus it inevitably contains various bugs similar to other software systems. These bugs can affect not only runtime efficiency but also the safety of code execution. Listing 1 shows the test case that can trigger a JIT compiler bug reported by our tool and is confirmed by the developers. This bug is severe and would crash the JVMs of OpenJDK 11, 17 and 19 [7]. Triggering this bug is non-trivial since it requires a test case and the specific values for certain options. In the test case of Listing 1, the optimization of accessing the elements of an array (i.e., `arr`) is buggy when unrolling the nested *for* loops (Line 4-8). However, the default value of the option `-Xcomp -XX:LoopUnrollLimit` is 50, which indicates it will unroll loop bodies with the node count (i.e., defined by the C2 compiler IR Graph [15]) less than 50. This option makes the test case unable to trigger the buggy optimization process. On the contrary, when setting the option `-Xcomp -XX:LoopUnrollLimit=500`, the test case can expose this bug since the loop unroll limit is set to 500.

```
1  public class Test {
2    public static void main(String[] strArr) {
3      int arr[] = new int[100];
4      for (int x = 0; x < 50; x++) {
5        for (int y = 1; y < 5; y++) {
6          // JVM incorrectly unrolls for loop
7          arr[x] -= 1;
8          arr[0] -= 1;
9  }}}}
```
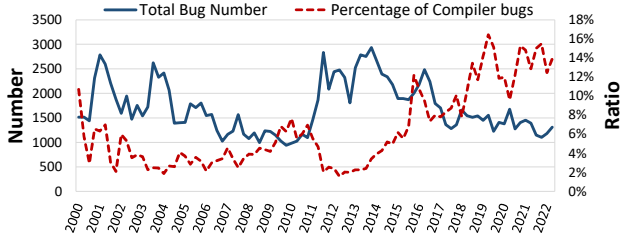
Listing 1: The Test Case to Trigger JDK-8284879

Fig. 1: The Evolution of the Total Number of Bugs and the Percentage of JIT Compiler Bugs (counted by every quarter)

Fig. 2: The Bug Priority Assigned to Bugs of Different Components. 'All' denotes the average priority of all components while 'Other' denotes the rest of the components.

Fig. 3: The Number of JDK Versions Affected by the Major Components of HotSpot. 'All' denotes all components of HotSpot. The horizontal line in the middle of the bar denotes the standard deviation. A rectangle filled with the pattern ⬚ indicates that the value of JIT compiler is significantly higher.

JIT compiler bugs have become increasingly pervasive over recent years. Specifically, Fig. 1 shows the total number of reported bugs in HotSpot [2] and the ratio of JIT compiler bugs (counted by every quarter) for OpenJDK over the last 20 years. We determine whether a bug belongs to the JIT compiler based on the labels of *component/subcomponent* of the bug report. It shows that the total number of bugs has decreased year by year over the last decade, while the proportion of JIT compiler-related bugs is increasing. In most recent years, the proportion has exceeded 16% in certain quarters. This trend shows that the JVM has become more functionally correct over the decades. However, developers increasingly demand the correctness and performance of optimization, while the implementation of optimization often contains various bugs.

The significance of JIT bugs can also be reflected by the priority assigned to the bugs. Usually, each bug report will be assigned with a *priority* by the developer with a severity level from P1 to P5 (i.e., from the highest to the lowest) [16]. Bugs with higher priorities are often more critical and require more urgent remediation. We investigate the priority distribution of different components in HotSpot, including over 26,243 reported bugs and 9,252 JIT compiler-related ones (see Section III for the details on the collection of such bugs). In particular, we select the six components with the highest number of bugs, including JIT Compiler, Runtime, GC, JFR, SVC, and JVMTI. Fig. 2 shows the statistical results. We can observe that JIT compiler contains 8.5% of P1 level bugs and the ratio of bugs in P1+P2+P3 exceeds 60%, which is the highest among all the major components. Such results reflect that developers often consider bugs in the JIT Compiler to be more severe and important than other components. We also observe that JIT compiler bugs often affect more JDK versions. Specifically, we investigate the number of JDK versions affected by the major components' bugs of HotSpot, and Fig. 3 shows the results. There are usually many versions of JDK, including the *long term support* (LTS) versions such
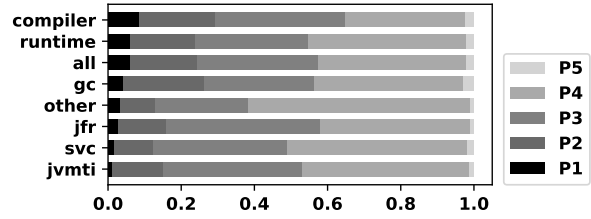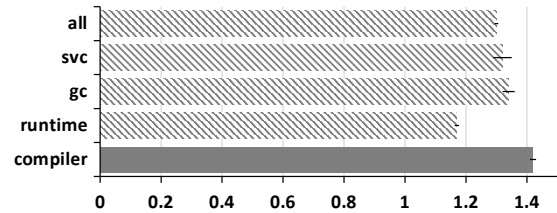
as JDK 8, 11, 17 and other Non-LTS versions such as JDK 9, 10, 12, etc. The more versions a bug affects, the more significant the bug is due to its widespread impact. The statistical results as shown in Fig. 3 indicate that JIT compiler bugs affect more JVM versions on average, indicating that such bugs are relatively more critical. Therefore, detecting JVM JIT compiler bugs is important.

C. Challenges

Effective fuzzing of JIT compilers and discovering optimization bugs are challenging. First, *merely exploring the dimension of generating new seeds is ineffective in triggering JIT compiler bugs* (Challenge§1). According to our investigation, nearly 60% of the existing JIT compiler bugs require specifying extra options other than the default ones (see Section III-B). However, existing JVM fuzzing techniques [8]–[10], [17] mainly focus on generating new seed inputs. Particularly, *classfuzz* [8] employs predefined mutation operators to mutate seeds at the syntactic level, such as changing variables' types and deleting exception handlers. *classming* [9] manipulates the control and data-flow of the seeds (e.g., insert or delete goto/return statements), which can generate simple *loop* structures. *JavaTailor* [10] generates new seeds by extracting code ingredients from historical test programs and randomly inserting these ingredients into the seed programs. Many generated seeds are invalid due to the lack of data dependency [9], let alone triggering deep optimization bugs. Even if the generated mutants can trigger optimizations, certain codes and logics cannot be explored when some options are disabled. Therefore, exploring the dimension of options is necessary.

Second, *exploring seeds and options collectively will significantly enlarge the search space, thus compromising both the effectiveness and efficiency of existing fuzzing techniques* (Challenge§2). As indicated by the example in Listing 1, certain optimization bugs can only be triggered when using specific code features and setting certain options to specific values. However, there are plenty of options (i.e., 211 options in HotSpot) in various types, such as Integer and String. Considering the possible values of various options and the diversity of code features, the search space will be explosive. In addition, there is no reliable information to guide the exploration of the large search space. As reported by existing studies [9], [10], the coverage statistics adopted by conventional fuzzing techniques are unsuitable for guiding JVM testing due to the non-determinism at runtime. Specifically, the optimizations performed under a particular configuration can vary since the code can be compiled in parallel by the JIT compiler, and the garbage collection can be performed as required. Hence, the coverage statistics on the JVM systems cannot be leveraged as effective guidance for fuzzing [10].

III. PRELIMINARY STUDY

To understand the characteristics of JIT compiler bugs, we first perform a comprehensive investigation of existing JIT compiler bugs. In particular, we performed our study on **HotSpot** for two main reasons. First, **HotSpot** is the most widely-used and high-performance virtual machine. In 2022, more than 90% of JDK distributions use HotSpot or HotSpot-based virtual machines [18]. Second, all the bug reports of HotSpot over the last decades have been well tracked in the JDK bug system (JBS) [19]. On the contrary, in other JVM implementations, bug reports are maintained in a lax manner. Specifically, we extracted 25,606 previously reported bugs of HotSpot, and 9,252 of them are JIT compiler related based on the label of *component* in the bug reports. Other popular components include Garbage Collection (GC), Runtime, Java Flight Recorder (JFR), Serviceability and etc.

This investigation aims to understand how JIT compiler bugs are triggered in practice. We are interested in understanding the characteristics and conditions (i.e., required options) of the triggering test cases. Besides, we aim to look for effective information that can guide the JIT compiler testing. Such results can shed lights on how to design more effective testing tools to detect JIT compiler bugs. To achieve these goals, we first extract the test cases from existing bug reports. We observe that developers typically use regression tests as triggering test cases in most bug reports. Such inputs cannot serve our purpose since a regression test often contains the test logic for multiple bugs. Therefore, we focus on extracting individual test inputs from bug reports' descriptions or attachments. In contrast to regression tests, such collected inputs can be executed independently. More importantly, they only correlate with specific issues, which can better reflect the characteristics of the corresponding bugs. Finally, we extract 955 such bug-triggering test cases in total from existing bug reports, with 540 of them belonging to the JIT compiler component.

TABLE I: The Code Structures of Bug Triggering Test Cases for JIT Compiler Bugs and Other Major Components.

Category	JIT Compiler	Runtime	GC	All
Assignment Stmt	**<u>2.48</u>**	0.38	0.83	1.61
If Stmt	**<u>0.35</u>**	0.22	0.23	0.29
Invocation	1.51	**2.75**	2.71	2.0
Arithmetic Operator	**<u>4.28</u>**	1.36	1.40	3.0
Shift Operator	**<u>0.11</u>**	0.03	0.02	0.08
Logical Operator	0.07	0.04	**0.09**	0.06
Unary Operator	0.69	0.27	**0.73**	0.53
Loop$_1$	0.56	0.32	**0.96**	0.50
Loop$_2$	**<u>0.29</u>**	0.05	0.16	0.19
Loop$_{3+}$	**<u>0.11</u>**	0.00	0.04	0.07
Switch	**<u>0.04</u>**	0.00	0.00	0.02
Try-Catch	0.12	**0.25**	0.14	0.17
Array	0.84	0.68	**0.87**	0.78
Field Access	**<u>4.08</u>**	1.40	1.96	2.97
Lambda	0.01	0.03	**0.07**	0.02
Synchronized Access	0.01	**<u>0.03</u>**	0.00	0.01

Array refers to the number of array operations (read/write/allocation). Loop$_n$ refers to the loop whose depth is n and 3+ refers to 3 or more. The bolded value indicates the maximum value and the underline denotes the value is significantly ($p < 0.05$) higher than others.

A. Code features of Triggering Test Cases

We first investigate the distribution of different code features in the triggering test cases of JIT compiler bugs and compare it to the other types of bugs. Specifically, we use *Spoon* [20] to analyze each triggering test case and count the number of code features, including *Language Features* and *Structure Features* as adopted by the existing study [21].

The results are shown in Table I. We can observe that JIT compiler bugs contain a significantly larger number of `assignment` statements and `Arithmetic` operators compared to that of Runtime and Garbage Collection. This is intuitive, since assignment statements contain most expressions and can have an impact on the data flow of the program. Thus, test cases related to data flow optimization (e.g., Conditional Constant Propagation) will contain more assignment statements. Besides, JIT compiler includes plenty of arithmetic optimizations (e.g., Peephole) concerning `arithmetic` operators. Operator-related bugs are often too subtle to be perceived by developers. Besides, test cases containing deeper `loops` are more likely to trigger JIT compiler bugs. While single-level loops can also trigger JIT optimizations, deeper ones have more complex logical relationships in the IR graph, making it more difficult to perform optimizations such as loop vectorization. In addition, array-related statements are used more often in the bug-triggering test cases of the JIT compiler. Compared with other types of variables, `Field` is often involved in the existence of variable escapes and memory value tracking, which is related to memory-related optimization strategies (e.g., stack allocation and scalar replacement). Since such features are typical subjects for optimization, they are more likely to trigger optimization errors.

[Finding-1] *To trigger JIT compiler bugs, the input class files usually contain a larger number of specific code features such as* `assign` *statement,* `field` *access,* `loop` *statement and* `arithmetic` *operator.*
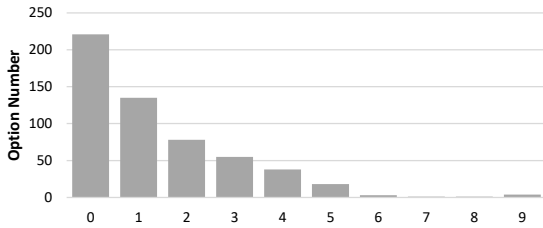
Fig. 4: The Distribution of Option Number

```
@ 18 java.lang.String::hashCode (49 bytes) callee is too large

@ 18 java.lang.String::hashCode (49 bytes) inline
  @ 19 java.lang.String::isLatin1 (19 bytes) inline
  @ 29 java.lang.StringLatin1::hashCode (42 bytes) inline
  @ 39 java.lang.StringUTF16::hashCode (not loaded) not inlineable
```

Fig. 5: The Profile Data of Inline.

B. Optimization Options

In addition to the required class files as inputs, we also observe that, 61.7% (333/540) of the JIT compiler bugs can only be triggered by the options with non-default values. We further investigate the distribution of the number of the options that require non-default values, and Fig. 4 shows the results. It can be seen that 59.5% (198/333) of such bugs require more than one option to be set. Such facts reveal new challenges for detecting JIT compiler bugs since the existing techniques [8]–[10], which merely mutate and generate class files, are incapable of discovering such bugs. Therefore, it motivates us to perform manipulations on the **configuration** (a sequence of options) when testing with newly generated inputs, which is a new dimension in JVM fuzzing. Unfortunately, exploring two dimensions (i.e., mutating seed files and manipulating configurations) collectively will enlarge the search space significantly. Particularly, we also observe that the options involved in the above test cases are diverse (i.e., only four options appeared over 20 times among different test cases). Such results further indicate that no option can trigger bugs stably. Consequently, random searching without guidance is highly likely to compromise both effectiveness and efficiency. Fortunately, we observe that some options can only affect those seeds with certain code features. As shown in Listing 1, the option `-XX:LoopUnrollLimit` is utilized to unroll loop bodies with the node count less than the specified number. Therefore, if there are no `loop` structures in the seed, testing the seed under such an option will bring no benefits. This motivates us to further learn about the correlations between options and various code features.

[**Finding-2**] *To trigger the majority of the JIT compiler bugs, certain optimizations are required to be set.*

C. Profile Data Difference

JVM typically collects the **profile data** to characterize and exploit dynamic program behaviors during program executions [22], [23]. In particular, the profile data contains the dynamic information about program execution (e.g., branch execution times, method call relationship) and JVM states at runtime (e.g., garbage collection information, CPU elapsed time of a method). Such profile data can significantly improve the quality of the optimized code and the efficiency of the optimization process [24]–[26]. To facilitate the analysis and performance optimization, JVM also provides certain flags for developers to access the profile data. For example, users can specify flag 'PrintAssembly' to output the optimized
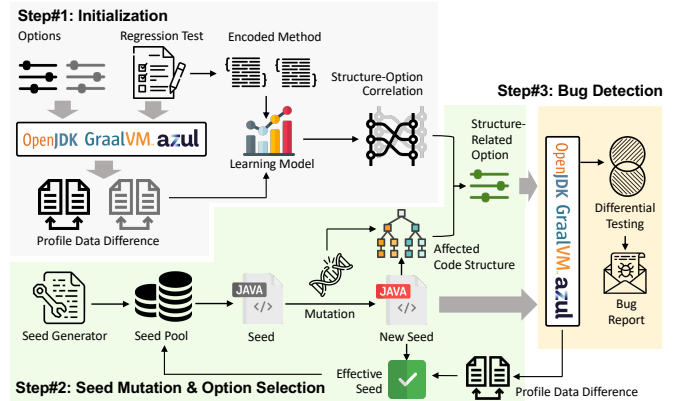

Fig. 6: Overview of the `JOpFuzzer`

assembly code and flag 'PrintBlockElimination' to print the progress when eliminating unnecessary basic blocks.

The profile data typically contains a list of log information, and each line indicates an optimization action. Fig. 5 shows the content of the profile data corresponding to inline optimizations with the option 'MaxInlineSize' set to different values. Each line represents an inline action, where the compiler tries to inline the function to the one in the previous line. The byte size of the function is indicated in parentheses. When the inlining succeeds, "inline" will be appended at the end of the line; otherwise, the failing reason will be logged. In Listing 5, the upper and below cases show the inlining results under the option value of "MaxInlineSize" set to 35 and 50, respectively. Specifically, when 'MaxInlineSize' is set to a larger value (the below case), the function `java.lang.String::hashCode` can inline the following three functions. Obtaining such profile data can help understand whether an optimization strategy has been applied and to what extent the code has been optimized.

[**Finding-3**] *The profile data is useful for manifesting how the code has been optimized at runtime.*

IV. APPROACH

In this work, we propose `JOpFuzzer` to test JVM JIT compilers, which focuses on searching for **effective test seeds** collectively with a set of **configurations** (*a sequence of options*) that are more likely to trigger optimization bugs. The core insights of `JOpFuzzer` are: (1) many JIT compiler bugs can only be triggered by seeds with certain code features under specific configurations collectively. Therefore, besides generating seeds, searching for diverse configurations along the new dimension of options is necessary; (2) since various optimizations will often break the integrity of code features and the optimized code with larger syntactic differences is

more likely to exhibit irregularities [27], thus triggering potential issues. Therefore, the profile data can serve as important information to guide the fuzzing process; and (3) the difference between the profile data can reveal to what extent the seeds have been optimized. A larger difference often indicates that the specified options are more sensitive to the seeds, particularly the enclosed code features. Therefore, it can be leveraged to learn the relationships between options and code features, thus eliminating the search space explosion problem. Based on the above insights, we proposed `JOpFuzzer` in this study, and the overview of which is shown in Fig. 6. Specifically, `JOpFuzzer` mainly contains three steps: **initialization**, including prior knowledge construction and initializing options; **seed mutation & configuration exploration**, which aims at generating new seeds and searching for correlated options guided by prior knowledge; and **bug detection**. The following presents the details of the three steps.

A. Initialization

In this step, we first manually collected all the options and initialized them according to the range of option values as recommended by developers. Specifically, we collected 79 options that are related to the mainstream optimization strategies. Meanwhile, we excluded certain specific options: the options without the corresponding profile data (e.g., Install-Methods); *verification options* that are used to verify optimizations (e.g., VerifyConnectionGraph); *stress options* that are used to stress the optimizations (StressLinearScan), and *broken options* confirmed by developers (e.g., TwoOperandLIRForm). For Boolean options, we use 0/1 to represent the disabling/enabling of the option. For numeric options, we choose the upper and lower boundary values as the candidate values, aiming to fully test the boundary conditions. For example, the option `EliminateAllocationArraySizeLimit` sets the size of the array to be optimized in the scalar replacement. The range of this option is from 0 to *Integer.MAX_VALUE*. Therefore, we choose 0 and *Integer.MAX_VALUE* as the candidates to control the impact of scalar replacement on arrays. This process is important since options should be initialized appropriately to avoid abnormal and invalid values.

After appropriate initialization, another important goal of this step is to construct the relationship between code features and options. Such relations can be utilized to effectively identify related options to trigger JIT optimizations when mutating specific code features, thus mitigating the search space explosion problem. To achieve this goal, we utilize the difference between the profile data obtained by specifying the designated options with different values to measure the effects. Our insight is that such effects can help us find the options that are highly likely to trigger JIT optimizations when we perform mutations on specific code features. For example, suppose we mutate the loop structures, we can select the correlated options to trigger *unroll loop optimization* instead of searching options blindly from the large search space. To construct such prior knowledge, we perform the following experiment before fuzzing on the regression tests of OpenJDK.

We choose regression tests since they are written by developers and contain various bug-triggering code features. To qualify the relationship between specific code features and options, we take a list of various options, a list of concerned code features, and the regression tests as inputs to generate the correlation matrix. The element of the matrix measures the correlation between a specific type of code feature and a specific option. In particular, we extract and encode the following code features following an existing study [21].

- **Language Features**. The language features include *Statements* and *Operators*, which are the basic and vital elements of Java language. Statements denote the set of Java statement types (e.g., *If* Stmt) and operators denote the set of all operator types (e.g., *Unary* Operators).
- **Structure Features**. The structure features focus on the existence of control-flow or data-flow related structural features. In particular, we consider: 1) the existence of array access and array elements of certain specific dimensions; 2) the existence of field access (usually requires the access to the class structures); 3) the existence of lambda expressions; 4) the existence of a loop with a specific depth; 5) the existence of the synchronized blocks (e.g., code block, method block, class block); and 6) the existence of the try-catch blocks.

We adopt one-hot encoding to represent code features. In particular, if a code feature exists in a method, the value of the corresponding feature is set to 1; otherwise, it is set to 0. To effectively learn the correlations, we keep only one option on and turn the others off at each time. In this study, for a non-boolean option, turning on means it is set to the upper boundary value while turning off to the lower boundary.

Then, we try to compile the code under the designated option with or without the selected method, and obtain the corresponding profile data separately. We work at the method level since it is the finest unit we can obtain the profile information through specifying options. In particular, we can set a compile command option to exclude the method in JIT compilation by specifying `-XX:CompileCommand=exclude,method`. If the profile data changes due to the inclusion/exclusion of a method, we can measure how the code features in this method are affected by the selected option. For example, suppose a method contains several code features and the corresponding profile data changes dynamically after the method is excluded in the JIT compilation under a specific option. We can then infer that part of the code features are likely to affect the option. Since the profile data is usually large and different types of profile data contain various features, we employ an efficient yet effective algorithm (i.e., the List Edit Distance (LED) algorithm [28]) to compare the difference between two profile data. In this study, we regard the similarity between two lines as 1 if they are the same and 0 otherwise. The intuition is to investigate how many edit operations are required to transfer one list into another. The higher operations required, the higher effect of the code features on the option. Specifically, the effect

between two profile data is calculated as follows:

$$getCorrelation(P_d, P_o) = \frac{LED(P_d, P_o)}{max(len(P_d), len(P_o))} \quad (1)$$

where P_d refers to the default profile data, P_o refers to the profile data when the method is excluded from the JIT compilation. $len(P_d)$ and $len(P_o)$ refer to the length of the two profile data, respectively. LED refers to the List Edit Distance algorithm [28]. We accumulate the correlation between a type of code feature and an option measured by traversing all the methods in the test suites. The matrix $\mathcal{M}_{access}$ is utilized to record the access numbers of such relations. Finally, the values in matrix $\mathcal{M}$ divided by the corresponding values in $\mathcal{M}_{access}$ are used to denote the final correlation.

Be noted that we utilize 25 out of the 85 different types of profile data that are related to optimization options (e.g., `PrintEliminateLocks` and `PrintInlining`). We exclude profile data that prints general information (e.g., `PrintCompilation` and `PrintClassHistogram`) or is not related to options. Besides, to minimize the non-deterministic effects of JIT, we specified the option `Xbatch`, which will proceed the compilation of all methods in a foreground task until completed, making the execution predictable and reproducible.

B. Seed Mutation & Configuration Exploration

In this section, we aim to collectively explore seed mutation and configuration selection to detect JVM JIT bugs efficiently. We detail the following steps: (1) generating a seed pool in Section IV-B1, (2) seed mutation in Section IV-B2, (3) configuration exploration for testing the mutated seeds in Section IV-B3 and (4) seed scheduling in Section IV-B4.

1) Seed Pool Generation: We obtain an initial seed pool using JavaFuzzer [29], a syntax-directed random generator of Java programs implemented by OpenJDK developers. We choose this tool since the Java programs generated by Java-Fuzzer can cover a wide range of syntax characteristics such as class inheritance, complex loop patterns, and enhanced exception-throwing patterns.

`JOpFuzzer` aims to prioritize and select seeds that are more likely to trigger JIT compiler bugs inspired by Finding-2 in Section III-A. Specifically, it prefers seeds with more `field access` statements and `arithmetic` operations to facilitate bug discovery since such features occur more often in existing JIT compiler's bug triggering inputs. It also prefers `loop` structures (including all depth) since the statements within loop bodies are executed repeatedly, and thus it is more likely to trigger JIT optimizations. Therefore, we sum up the number of these three features for all the seeds and sort them in the descending order.

2) Seed Mutation: According to our preliminary study, not every code feature can trigger the JIT optimization process. Therefore, randomly selecting a code feature for mutation offers limited benefits. Similar to the previous step, we also prefer to mutate the code features that are more likely to trigger

TABLE II: Code Change Patterns Adopted by TBar

| Pattern	Pattern
Insert Cast Checker	Insert Null Pointer Checker
Insert Range Checker	Insert Missed Statement
Mutate Class Instance Creation	Mutate Conditional Expression
Mutate Data Type	Mutate Integer Division
Mutate Method Invocation	Mutate Literal Expression
Mutate Operators	Mutate Return Statement
Mutate Variable	Move Statement
Remove Statement	
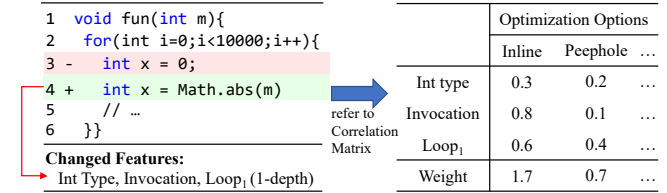
Fig. 7: Illustrations of How `JOpFuzzer` Mutates Seeds, Extracts the Changed Features and Computes the Weights of Optimization Options.

JIT optimizations. In particular, we assign the weight $w(s)$ to each statement s according to the following equation:

$$w(s) = inLoop(s) + hasField(s) + hasArithOp(s) \quad (2)$$

where $inLoop$ examines whether s resides in a loop body, $hasField$ examines whether s contains field access and $hasArithOp$ examines whether s contains arithmetic operators. Their value is 1 if s contains the corresponding code feature and 0 otherwise. We then select one statement to mutate by the following potential function:

$$potential(s_i) = \frac{w(s_i)}{\sum_{i=1}^{n} w(s_i)} \quad (3)$$

where s_i is the ith statement in the seed, n is the total number of the statements. Intuitively, the higher a statement's potential, the more likely it will be mutated.

After selecting the mutation point (i.e., a statement), `JOpFuzzer` aims to perform mutation on it by altering the semantic logic of existing seeds to generate new ones. It deliberately destroys the data-flow or control-flow of a seed to generate corner cases, expecting to trigger more optimization behaviors. Our insight is that the mutation may alter the data dependencies (the define-use chain of variables) or the path constraints that pass through the mutation point, which may result in abnormal execution behaviors of the seed. Therefore, the more semantic logic is disrupted, the more likely the mutant will cause the JVM to fail when it performs sophisticated optimizations. In this study, we apply the code change patterns as adopted by TBar [30] to perform the mutation. TBar is a template-based approach that can be used to mutate Java programs originally designed for automated program repair [30]–[33]. We choose TBar since it can offer flexible and diverse mutation patterns for a given statement. Specifically, TBar offers 15 categories of mutation patterns and contains 35 mutation templates, such as variable replacing and mutating conditional expressions. Table II summarizes the mutator utilized by TBar [30].

3) Configuration Exploration: `JOpFuzzer` tends to select a mutation point that contains optimization-triggering code features and performs mutations with pre-defined mutators. In this step, we demonstrate how to select a **configuration** (i.e., a sequence of options) to trigger JIT optimizations on the mutants, and Fig. 7 shows the overall idea. Intuitively, setting the configuration should consider both the context of the mutation point and the mutated code features. The context can offer a holistic view of the code features that can be utilized to trigger more optimizations, such as loop unrolling. The mutated statements may contain specific features triggering more microscopic optimization behaviors, such as peephole optimization. Therefore, `JOpFuzzer` extracts the code features that contain the mutation point (e.g., $Loop_1$) and the code features within the mutated statements. We denote such code features as the *changed features* as shown in Fig. 7. Then, we refer to the correlation matrix (as introduced in Section IV-A) to obtain the configurations that maximize the probability of triggering possible optimizations for the changed features.

In our design, the higher the correlation value, the more likely an option can trigger possible optimizations for the code features. Specifically, suppose there are n changed code features $L_c = \{L_1, L_2, \ldots, L_n\}$ where L_i denotes one of the code features, and an option o. We query about the correlation matrix M and sum up the correlation value of o on all the changed features as the weight for including option o in the final configuration. Fig. 7 shows an example of the process, in which `JOpFuzzer` mutates the seed, extracts the changed features and calculates the weight for each option. Then, for all the m options $L_{op} = \{o_1, o_2, \ldots, o_m\}$, we use $weight(o_i)$ to denote the weight for the option o_i and we include an option in the configuration by the following potential function:

$$potential(o_i) = \frac{weight(o_i)}{\sum_{i=1}^{m} weight(o_i)} \quad (4)$$

Accordingly, the higher potential for an option, the higher probability for `JOpFuzzer` to select it. `JOpFuzzer` will select three options as the final configuration to test the mutant. We limit the size to three since over 80% (268/333) of the triggering test cases involve three or fewer options accordingly to our preliminary study as shown in Fig. 4. Besides, we select such a number to balance efficiency and effectiveness. In particular, we will enumerate all the possible values of the selected options for differential testing since certain bugs can be triggered only when turning some options on while others off. For example, triggering JDK-8286871 needs to turn on `OptoNoExecute` while turning off `TLABStats`. Such enumeration will result in a large search space, and thus increasing the configuration size will significantly increase `JOpFuzzer`' overheads, thus compromising its efficiency.

4) Seed scheduling: `JOpFuzzer` prefers the mutants that can trigger the optimization and hence it will keep such effective mutants in the seed pool. As the profile data can reflect the optimization process, we utilize the difference between the profile data before and after specifying options to measure the effectiveness of a mutant. Specifically, given a mutant and a configuration ($size = 3$), for each option o, we can obtain two profile data by turning on and off the option when executing the mutant. Then we utilize Equation 1 to calculate the correlation between the two profile data under option o. This way, we can obtain three correlation values for the configuration. We consider a mutant is effective if all of the three correlation values exceed the average correlation value among the correlation matrix since such a value measures the average effect of the options on various code features. Finally, `JOpFuzzer` will append the effective mutant to the seed pool for further testing. Otherwise, the mutants will be abandoned.

C. Bug Detection

`JOpFuzzer` mainly employs the following two test oracles to detect bugs:

Crash: If JVM crashes at runtime, a crashing bug is considered to be triggered. The root cause of the crash can be checked and analyzed in an automatically generated file named `hs_err_pid.log`.

Inconsistent: We check whether the target JVM outputs inconsistent results for the same input under different optimization configurations. However, inconsistent results do not necessarily indicate real bugs since some optimization will output extra information, such as certain warning messages generated for users. Therefore, we further perform manual investigation to identify bugs to check whether such inconsistency is indeed caused by an optimization error.

V. EVALUATION

We performed experiments with the aim to evaluate the effectiveness and usefulness of `JOpFuzzer` via answering the following research questions:

- **RQ1 (Effectiveness)**: How effective is `JOpFuzzer` in terms of code coverage and bug detection?
- **RQ2 (Components' Contribution)**: How is the contribution of the major components of `JOpFuzzer`?
- **RQ3 (Usefulness)**: Can `JOpFuzzer` detect real JVM JIT compiler bugs?

A. Evaluation Setup

Target JVMs. We select the most popular JVM implementation of HotSpot [34] as our test target. We exclude OpenJ9 since there are few flags available for users to obtain the profile data. Specifically, we choose the latest debug build of OpenJDK8, OpenJDK11, OpenJDK17 in our experiments (i.e., build 25.71-b00, build 11.0.15 and build 17.0.3). We compiled the latest debug build of these JDKs since the product versions will not crash directly when encountering optimization bugs by rolling back to interpretation execution. On the contrary, the debug build will add `assert` statements to help developers check the optimization states during execution.

Baseline. We compare `JOpFuzzer` with two state-of-the-art JVM testing technologies *JavaTailor* [10] and *class-ming* [9], which have been introduced in Section II-C. For

TABLE III: Coverage Achieved by JavaFuzzer, classming, JavaTailor and `JOpFuzzer`

Category	OpenJDK11		OpenJDK17	
	Line	Function	Line	Function
JavaFuzzer	43.9	38.9	45.1	39.4
classming	42.1	38.1	44.1	39.0
JavaTailor	46.7	40.8	47.3	41.1
JOpFuzzer	**56.2**	**47.6**	**57.5**	**47.9**

fair comparison, we use the same seed pool generated by *Java-Fuzzer* as the seed pool for *JavaTailor* and *classming*. Since *classming* iteratively mutates seed programs, it is required to set a number of iterations on each seed program. Following the experiment settings in the previous work [9], [10], the average number of iterations for a program in *classming* is 14 times of the number of the program instructions. Therefore, we use the same number to limit the number of iterations.

Environment. Our evaluation was conducted on a linux server with Intel(R) Xeon(R) Gold 6248R CPU and 256GB RAM. To collect the coverage of JVM, we adopt the coverage tool GCOV and LCOV to collect the coverage statistics [35]. Noted that OpenJDK8 does not support the configuration option `--enable-native-coverage` which enables native compilation with code coverage data. Thus, we did not report the coverage results on OpenJDK8. In particular, we focus on the coverage statistics of HotSpot and ignore other JDK components.

B. RQ1: Effectiveness of `JOpFuzzer`

In this RQ, we compare `JOpFuzzer` with *JavaTailor* and *classming* on OpenJDK11 and OpenJDK17 with respect to the achieved code coverage and the number of detected bugs. Besides, we also collect the initial seed coverage (i.e., denoted as JavaFuzzer) to reveal the improvement in coverage by different tools. We run each tool on generated test programs for 24 hours and repeat the experiments three times.

Code Coverage. Table III shows the results in terms of code coverage, which is averaged over the repeated experiments. We can observe that `JOpFuzzer` outperforms the two baselines in terms of both line and function coverage. Specifically, `JOpFuzzer` achieves 56.2%, 47.6% on the line and function coverage for OpenJDK11 as well as 57.5%, 47.9% for OpenJDK17. On average, `JOpFuzzer` outperforms the best baseline by 20.3% and 16.7% for the line and function coverage on OpenJDK11, respectively. As for OpenJDK17, `JOpFuzzer` outperforms the best baselines for 21.6% and 16.5% for the line and function coverage. Note that OpenJDK11 and OpenJDK17 consist of about 620KLoc and 656KLoc, respectively. Hence, even 1% coverage improvement can result in thousands of lines being covered.

JavaTailor can only achieve the line coverage of 46.7% and 40.8% for function coverage for OpenJDK11. We found that the seed survival rate of *JavaTailor* was relatively low after mutation despite the recovery of broken constraints. For seeds with more complex control and data flows generated by *JavaFuzzer*, the seed survival rate is only about 30%. This suggests that *JavaTailor* inadequately fixes those constraints

with complex data flow dependencies. Regarding *classming*, it achieves 42.1% and 38.1% in terms of the line coverage and function coverage on OpenJDK11. The corresponding values are 44.1% and 39.0% on OpenJDK17. The coverage achieved by *classming* is even lower than that of the original seeds since the mutation is inefficient and not scalable. In particular, *classming* requires a sequence of executed bytecode instructions (i.e., live bytecode) during mutation. Relying on such live bytecode, *classming* selects the position with the most interceptions of data dependencies to insert jump statements in each function. However, the target of JIT optimization is repetitive code snippets (i.e., HotSpot code), which often contain plenty of loops and function calls. Therefore, the number of live bytecode recorded by *classming* is explosive while the data dependency analysis is extremely time-consuming. Therefore, its efficiency has been significantly compromised.
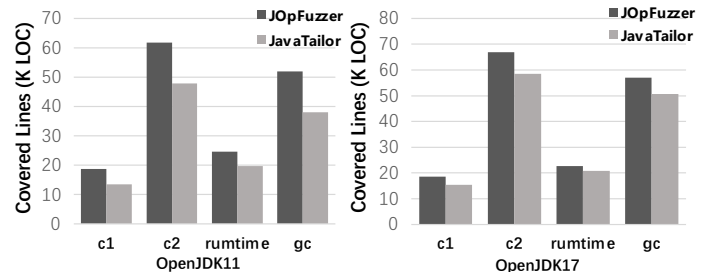

Fig. 8: Line Coverage Comparison in the Major Components

We further investigate the improvement of line coverage by analyzing how many lines are covered for the major components compared to the best baseline *JavaTailor* with the same experiment setting. Fig. 8 shows the statistical results. We can observe that `JOpFuzzer` boosts the line coverage on the C2 compiler and garbage collection significantly. We presume this is because `JOpFuzzer` forces the JVM to perform the corresponding optimizations during the configuration exploration, which requires more memory allocation, thus resulting in more lines being covered. Such results demonstrate that `JOpFuzzer` can improve not only the coverage of the JIT compiler but also the coverage of other components.

Bug Detection. Table IV shows the total number of detected bugs results. `JOpFuzzer` has detected three, six and four bugs in total on OpenJDK8, OpenJDK11 and OpenJDK17 respectively, while both *JavaTailor* and *classming* cannot detect any bugs within 24 hours. Besides the reasons as discussed above, another major reason behind the ineffective bug detection of *JavaTailor* and *classming* is that they mainly focus on detecting the differences between HotSpot and OpenJ9 while `JOpFuzzer` focuses on detecting the difference between the optimization strategies employed by different versions of HotSpot. Consequently, it is challenging for the existing tools to detect those optimization bugs in HotSpot.

C. RQ2: Components' Contribution

In this section, we focus on evaluating the key components of `JOpFuzzer`. We design the following three variants and compare with `JOpFuzzer` in terms of code coverage.

TABLE IV: Bugs Detected by JOpFuzzer, JavaTailor, Classming, JOpFuzzer-RandomOP and JavaTailor-OP.

Tools	OpenJDK8	OpenJDK11	OpenJDK17	Total
JOpFuzzer	3	6	4	13
JavaTailor	0	0	0	0
classming	0	0	0	0
JOpFuzzer$_{rop}$	1	2	1	4
JavaTailor$_{op}$	1	3	3	7

- **JOpFuzzer$_{rop}$**: this variant explores configuration by randomly selecting three options each time instead of being guided by our learned prior knowledge. Besides, it does not perform seed scheduling neither (see Section IV-B4).
- **JOpFuzzer$_{ns}$**: this variant explores configuration in the same way as JOpFuzzer, but it does not perform seed scheduling as introduced in Section IV-B4.
- **JavaTailor$_{op}$**: this variant equips *JavaTailor* with our configuration exploration strategies. In particular, we extract the changed features for the mutants generated by *JavaTailor* and select the correlated options to test the mutant. This variant can reveal whether our strategies can boost the performance of other mutation-based approaches.

Table V summarizes the code coverage achieved by the three variants on JDK11 and JDK17, respectively. The results show that each major component of JOpFuzzer contributes to its promising performance. Specifically, by removing the seed scheduling component, the line and function coverages have been decreased on both JDK versions (by comparing JOpFuzzer and JOpFuzzer$_{ns}$). The main contribution of this component is to eliminate those seeds that are insensitive to the configurations and hence improve the efficiency of the testing process. If we eliminate the configuration exploration strategies, the coverages achieved have decreased further. In particular, the function coverage has further been decreased by 5.3% ((45.5%-43.1%)/45.5%). Similar results are observed for line coverage as well as the results for other JDK versions. This indicates that our devised configuration exploration strategies contributed significantly to JOpFuzzer's performance.

We also made some other interesting findings. First, by randomly exploring optimization options, JOpFuzzer's performance can also outperform existing baselines (by comparing JOpFuzzer$_{rop}$ with JavaTailor). Such results are align our previous findings in the preliminary study, which reveal that merely exploring seed mutation is ineffective in detecting JVM JIT compiler bugs. Therefore, it confirms our motivation to perform two-dimensional input space exploration. Second, our devised configuration exploration can also boost the performance of existing state-of-the-art techniques (by comparing JavaTailor and JavaTailor$_{op}$). Such results further prove that our idea of learning correlations between code features and optimization options, and utilizing correlations to guide the fuzzing process is feasible and effective.

We further analyzed the bugs detected by the designed variants. We mainly compare JOpFuzzer with JOpFuzzer$_{rop}$ and JavaTailor$_{op}$ since the component of option selection makes the major contribution to JOpFuzzer. Table IV shows the total number, and we can see that JOpFuzzer$_{rop}$ and

TABLE V: Code Coverage Achieved by the Three Variants

Tools	OpenJDK11		OpenJDK17	
	Line	Function	Line	Function
JOpFuzzer$_{rop}$	52.1	43.1	52.7	43.5
JOpFuzzer$_{nos}$	54.6	45.5	55.1	45.8
JOpFuzzer	**56.2**	**47.6**	**57.5**	**47.9**
JavaTailor	46.7	40.8	47.3	41.1
JavaTailor$_{op}$	**53.4**	**45.6**	**53.8**	**45.2**
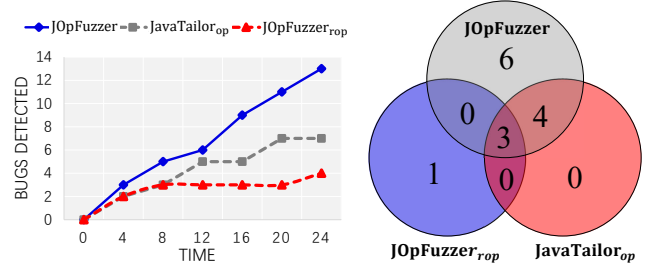
(a) The Number of Bugs Detected Over Time of Different Tools **(b)** Overlap between the Detected Bugs of Different Tools

Fig. 9: Dissecting the Number of Detected Bugs

JavaTailor$_{op}$ can detect four and seven bugs respectively, which outperforms existing baselines. Fig. 9a depicts the number of detected bugs over time for JOpFuzzer and the variants within 24 hours. We can see that JOpFuzzer can detect more bugs continuously over time. On the contrary, JavaTailor$_{op}$ can only detect limited number of bugs, which is attributed to two main reasons. First, it will introduce new loops and external function calls during the fuzzing process, compromising its efficiency. Second, due to the low survival rate of the generated seeds, the effectiveness is also degraded. We further analyzed the overlap among the bugs detected by different tools, and Fig. 9b shows the results. As it reveals, JOpFuzzer can detect almost all the bugs that are detected by the other variants. Only one missed case is detected by JOpFuzzer$_{rop}$ but not JOpFuzzer. This bug was caused under the option `PinAllInstructions` while this option does not correlate to any specific profile data. Therefore, JOpFuzzer excluded it and did not specify it during runtime. JOpFuzzer$_{rop}$ detected it since it randomly selects among all the optimization options.

D. RQ3: Usefulness of JOpFuzzer

Summary of the detected bugs. We applied JOpFuzzer to test the latest GitHub commits of OpenJDK8, OpenJDK11, OpenJDK17 and OpenJDK20. The first three are long-term supported versions of OpenJDK with the largest market share, and OpenJDK20 is the mainline version of OpenJDK. During the two months of testing, JOpFuzzer detected 41 bugs, with 25 confirmed or fixed. These bugs are related to the JIT compiler and require a specific configuration to be triggered, so they cannot be detected by either JavaTailor nor classming. We identified 33 distinct options among those confirmed bugs, and found that 88% of those bugs involve one or two options after reduction. Such options cover inline, loop unrolling, peephole and many other critical optimizations. Such bugs are

```
1  public class ClearArray {
2    static long[] STATIC;
3    static void foo() {STATIC = new long[2048 - 1];}
4    public static void main(String[] args) {
5      for (int i = 0; i < 20000; ++i) {
6        foo(); // crash the JVM
7             }}}
```

Listing 2: The Test Case to Trigger JDK-8284883

challenging to detect, which often require input with specific code structures with specific options.

Case Study. To demonstrate why `JOpFuzzer` can effectively expose JIT compiler bugs, we perform a case study as shown in Listing 2 [36]. Specifically, it shows the test case that triggers bug JDK-8284883 discovered by `JOpFuzzer` and Listing 2 shows the minimized version by the corresponding developers. The JVM will crash by specifying options such as `-Xcomp -XX:+UnlockDiagnosticVMOptions -XX:-IdealizeClearArrayNode`. The last option specified will trigger JIT optimizations when `array` exists (line 3).

VI. DISCUSSION

Threats to validity. This study suffers from the following main threats. First, our empirical study only involves instances from HotSpot. Therefore, the empirical findings might not generalize well to other JVM implementations. However, we collected large-scale bug reports over the last decades, including 26,243 previously reported bugs. Besides, HotSpot is widely used, and more than 90% of JDK distributions use HotSpot or HotSpot-based virtual machines [18]. Therefore, the generalizability issue is mitigated due to the representativeness of HotSpot. Another threat concerns the evaluations. For JavaTailor, we directly reuse the provided source code. For classming, we carefully re-implemented it according to the paper's description and checked its correctness by reproducing the original results to avoid potential bias. JOpFuzzer's effectiveness is affected by the selection of the initial seeds. For example, some bugs require multiple nested loops with special data types (e.g., high-dimensional arrays) to be triggered, and these code structures cannot be easily obtained through mutation. To mitigate the uncertainty introduced by the initial seed selection, we generated multiple sets of seeds and repeated our experiments three times.

Ethical Considerations. To facilitate developers debugging, we submit detailed information such as the bug-triggering options and error logs. We also perform bug reduction and deduplication to our best before reporting bugs. Besides, to avoid spamming the open-source community and the developers, we have carefully checked the reproducibility of bugs on at least one long-term-support JVM before we submit them. The developers have responded positively to our bug reports.

VII. RELATED WORK

JVM testing. Due to the importance of the JVM, much research has been proposed to generate effective bug-revealing test cases [8]–[10], [24], [37]–[39]. In addition to the aforementioned *classfuzz* [8], *classming* [9] and *JavaTailor* [10], there is plenty of work related to JVM testing.

Laurent *et al.* [40] extend an advanced mutation testing tool, PITest [41], to enable it to record full information about the mutant when the JVM crashes (e.g. failing and passing cases). Then the authors utilize such information to predict which features of the mutant are more likely to crash the JVM. Hwang *et al.* [39] proposed JUSTGen, which generates test cases to identify the unspecified behavior of JNI specification.

Optimization Exploration for Compilers. Much research has focused on exploring optimization options to trigger compiler bugs. Le *et al.* [42] proposed a randomized stress-testing technique for detecting optimization bugs of GCC and LLVM at link-time. Jiang *et al.* [43] proposed a technique CTOS to detect compiler bugs when applying arbitrary optimization sequences. CTOS aims to capture the information of optimization sequences and the testing program and detect the LLVM bugs by applying differential testing. Chen *et al.* [21] proposed a tool, COTest, to detect compiler bugs for LLVM and GCC. COTest adopts machine learning to model the relationship between test programs and options, aiming to predict the bug-triggering probability of a test program under specific options. COTest has detected previously unknown bugs of LLVM and GCC. However, we cannot use the above approaches as our baselines because they rely on specific features from compilers, which are unavailable from JVM. For example, Proteus aims to test link-time bugs; CTOS utilizes the IR generated by LLVM to capture the semantics of the testing program; COTest utilizes options unique to the GNU compiler (e.g., -O1,-O2,-O3). `JOpFuzzer` is the first study that combines options and Java code features to test JVM, which can inspire future JVM testing work.

VIII. CONCLUSION

In this work, we conducted a large-scale empirical study over 26,243 real bugs collected from OpenJDK to understand JVM JIT Compiler bugs. Our study reveals the significance of JIT compiler bugs. Moreover, we observed that seed inputs containing more specific code features are more likely to trigger JIT compiler bugs, and often require certain optimization configurations. Inspired by our empirical findings, we devise a novel fuzzing approach, named `JOpFuzzer`, to detect JIT compiler bugs by collectively exploring the two-dimensional spaces of seeds and options. Extensive evaluations demonstrate both the effectiveness and usefulness of `JOpFuzzer`.

IX. ACKNOWLEDGEMENT

REFERENCES

[1] "Language ranking," 2021. [Online]. Available: https://madnight.github.io/githut/#/pull_requests/2021/3

[2] "Hotspot," 2022. [Online]. Available: http://openjdk.java.net

[3] "Dragonwell," 2022. [Online]. Available: https://github.com/alibaba/dragonwell11

[4] "Openj9," 2022. [Online]. Available: https://www.eclipse.org/openj9.

[5] "Zulu," 2022. [Online]. Available: http://www.azulsystems.com/products/zulu.

[6] A. Sonoyama, T. Kamiyama, M. Oguchi, and S. Yamaguchi, "Performance study of kotlin and java program considering bytecode instructions and JVM JIT compiler," in *Ninth International Symposium on Computing and Networking, CANDAR 2021 - Workshops, Matsue, Japan, 23-26 November 2021*. IEEE, 2021, pp. 127–133. [Online]. Available: https://doi.org/10.1109/CANDARW53999.2021.00028

[7] "Jdk-8284879," 2022. [Online]. Available: https://bugs.openjdk.java.net/browse/JDK-8284879

[8] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of JVM implementations," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, C. Krintz and E. D. Berger, Eds. ACM, 2016, pp. 85–99. [Online]. Available: https://doi.org/10.1145/2908080.2908095

[9] 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.

[10] Y. Zhao, Z. Wang, J. Chen, M. Liu, M. Wu, Y. Zhang, and L. Zhang, "History-driven test program synthesis for JVM testing," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1133–1144. [Online]. Available: https://doi.org/10.1145/3510003.3510059

[11] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Q. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: an automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, A. C. Arpaci-Dusseau and G. Voelker, Eds. USENIX Association, 2018, pp. 578–594. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen

[12] M. Paleczny, C. A. Vick, and C. Click, "The java hotspot server compiler," in *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*, S. Wold, Ed. USENIX, 2001. [Online]. Available: http://www.usenix.org/publications/library/proceedings/jvm01/paleczny.html

[13] K. Shiv, R. Iyer, C. Newburn, J. Dahlstedt, M. Lagergren, and O. Lindholm, "Impact of JIT/JVM optimizations on java application performance," in *7th Annual Workshop on Interaction between Compilers and Computer Architecture (INTERACT-7 2003), 8 February 2003, Anaheim, CA, USA*. IEEE Computer Society, 2003, pp. 5–13. [Online]. Available: https://doi.org/10.1109/INTERA.2003.1192351

[14] T. Yoshikawa, K. Shimura, and T. Ozawa, "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*. IEEE Computer Society, 2003, p. 20. [Online]. Available: https://doi.org/10.1109/QSIC.2003.1319081

[15] "C2 ir graph and nodes," 2022. [Online]. Available: https://wiki.openjdk.java.net/display/HotSpot/C2+IR+Graph+and+Nodes

[16] "Jdk bug priority," 2021. [Online]. Available: https://wiki.openjdk.org/display/jmc/ILW+Prioritization

[17] E. G. Sirer and B. N. Bershad, "Using production grammars in software testing," in *Proceedings of the Second Conference on Domain-Specific Languages (DSL '99), Austin, Texas, USA, October 3-5, 1999*, T. Ball, Ed. ACM, 1999, pp. 1–13. [Online]. Available: https://doi.org/10.1145/331960.331965

[18] "2022 java developer productivity report," 2022. [Online]. Available: https://www.jrebel.com/resources/java-developer-productivity-report-2022

[19] "Jdk bug system," 2022. [Online]. Available: https://bugs.openjdk.java.net/secure/Dashboard.jspa

[20] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "SPOON: A library for implementing analyses and transformations of java source code," *Softw. Pract. Exp.*, vol. 46, no. 9, pp. 1155–1179, 2016. [Online]. Available: https://doi.org/10.1002/spe.2346

[21] J. Chen and C. Suo, "Boosting compiler testing via compiler optimization exploration," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, pp. 72:1–72:33, 2022. [Online]. Available: https://doi.org/10.1145/3508362

[22] A. Sarimbekov, A. Podzimek, L. Bulej, Y. Zheng, N. P. Ricci, and W. Binder, "Characteristics of dynamic JVM languages," in *VMIL@SPLASH '13: Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages, Indianapolis, IN, USA, 28 October 2013*, C. Bockisch, M. Haupt, S. Blackburn, H. Rajan, and J. Gil, Eds. ACM, 2013, pp. 11–20. [Online]. Available: https://doi.org/10.1145/2542142.2542144

[23] L. Zhang and C. Krintz, "Profile-driven code unloading for resource-constrained jvms," in *Proceedings of the 3rd International Symposium on Principles and Practice of Programming in Java, PPPJ 2004, Las Vegas, Nevada, USA, June 16-18, 2004*, ser. ACM International Conference Proceeding Series, J. Waldron, Ed., vol. 91. ACM, 2004, pp. 83–90. [Online]. Available: https://dl.acm.org/citation.cfm?id=1071581

[24] A. W. Wade, P. A. Kulkarni, and M. R. Jantz, "Exploring impact of profile data on code quality in the hotspot JVM," *ACM Trans. Embed. Comput. Syst.*, vol. 19, no. 6, pp. 48:1–48:26, 2020. [Online]. Available: https://doi.org/10.1145/3391894

[25] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani, "Design and evaluation of dynamic optimizations for a java just-in-time compiler," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 4, pp. 732–785, 2005. [Online]. Available: https://doi.org/10.1145/1075382.1075386

[26] A. W. Wade, P. A. Kulkarni, and M. R. Jantz, "AOT vs. JIT: impact of profile data on code quality," in *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2017, Barcelona, Spain, June 21-22, 2017*, V. Nagarajan and Z. Shao, Eds. ACM, 2017, pp. 1–10. [Online]. Available: https://doi.org/10.1145/3078633.3081037

[27] X. Ren, M. Ho, J. Ming, Y. Lei, and L. Li, "Unleashing the hidden power of compiler optimization on binary code difference: an empirical study," in *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, S. N. Freund and E. Yahav, Eds. ACM, 2021, pp. 142–157. [Online]. Available: https://doi.org/10.1145/3453483.3454035

[28] E. S. Ristad and P. N. Yianilos, "Learning string-edit distance," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 20, no. 5, pp. 522–532, 1998. [Online]. Available: https://doi.org/10.1109/34.682181

[29] "Javafuzzer test generator," 2022. [Online]. Available: https://github.com/shipilev/JavaFuzzer

[30] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, D. Zhang and A. Møller, Eds. ACM, 2019, pp. 31–42. [Online]. Available: https://doi.org/10.1145/3293882.3330577

[31] M. Wen, J. Chen, R. Wu, D. Hao, and S. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 1–11. [Online]. Available: https://doi.org/10.1145/3180155.3180233

[32] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin, "Automated patch correctness assessment: How far are we?" in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 968–980. [Online]. Available: https://doi.org/10.1145/3324884.3416590

[33] B. Lin, S. Wang, M. Wen, and X. Mao, "Context-aware code change embedding for better patch correctness assessment," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 3, may 2022. [Online]. Available: https://doi.org/10.1145/3505247

[34] "Openjdk8," 2022. [Online]. Available: https://github.com/openjdk/jdk8u

[35] G. Kondoh and T. Onodera, "Finding bugs in java native interface programs," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, B. G. Ryder and A. Zeller, Eds. ACM, 2008, pp. 109–118. [Online]. Available: https://doi.org/10.1145/1390630.1390645

[36] "Jdk-8284883," 2022. [Online]. Available: https://bugs.openjdk.java.net/browse/JDK-8284883

[37] T. Brennan, S. Saha, and T. Bultan, "JVM fuzzing for jit-induced side-channel detection," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 1011–1023. [Online]. Available: https://doi.org/10.1145/3377811.3380432

[38] J. Zhao, Y. Wen, X. Li, L. Pang, X. Kuang, and D. Wang, "A heuristic fuzz test generator for java native interface," in *Proceedings of the 2nd ACM SIGSOFT International Workshop on Software Qualities and Their Dependencies, SQUADE@ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26, 2019*, S. Sentilles, B. W. Boehm, C. Trubiani, and A. Koziolek, Eds. ACM, 2019, pp. 1–7. [Online]. Available: https://doi.org/10.1145/3340495.3342749

[39] S. Hwang, S. Lee, J. Kim, and S. Ryu, "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. IEEE, 2021, pp. 1708–1718. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00151

[40] T. Laurent, F. Wall, and A. Ventresque, "On the impact of timeouts and JVM crashes in pitest," in *13th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 2020, pp. 247–253. [Online]. Available: https://doi.org/10.1109/ICSTW50294.2020.00050

[41] "Pitest," 2022. [Online]. Available: http://pitest.org/

[42] V. Le, C. Sun, and Z. Su, "Randomized stress-testing of link-time optimizers," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, M. Young and T. Xie, Eds. ACM, 2015, pp. 327–337. [Online]. Available: https://doi.org/10.1145/2771783.2771785

[43] H. Jiang, Z. Zhou, Z. Ren, J. Zhang, and X. Li, "CTOS: compiler testing for optimization sequences of LLVM," *IEEE Trans. Software Eng.*, vol. 48, no. 7, pp. 2339–2358, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2021.3058671