Fuzzing Java Optimizing Compilers with Complex Inter-Class Structures Guided by Heterogeneous Program Graphs

Shiyu Qiu*†
Huazhong University of Science and Technology, Wuhan, China
qiusy@hust.edu.cn

Ming Wen*†‡
Huazhong University of Science and Technology, Wuhan, China
mwenaa@hust.edu.cn

Zifan Xie*†
Huazhong University of Science and Technology, Wuhan, China
xzf1244@gmail.com

Hai Jin*§
Huazhong University of Science and Technology, Wuhan, China
hjin@hust.edu.cn

Abstract

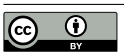
Modern Java compilers, such as *Just-In-Time* (JIT) and *Ahead-Of-Time* (AOT) compilers, often infer and analyze complex inter-class structures to perform program optimizations while preserving semantic correctness. However, incorrect inference of these structures during compilation can lead to critical bugs, as observed in production compilers like HotSpot and R8. Despite their effectiveness, existing Java fuzzing tools fail to adequately explore inter-class relationships, focusing instead on simpler intra-class or intra-method constructs. To bridge this gap, we present InterFuzz, the first fuzzing framework designed to systematically generate Java test cases with complex inter-class structures. InterFuzz introduces a novel concept of *Heterogeneous Program Graph* (HPG) to abstract and manipulate inter-class relationships. It then employs Inter-Class Mutators to construct intricate interactions, and utilizes graph complexities to guide test generation toward high-diversity code. Our evaluation shows that InterFuzz effectively uncovers compiler bugs that elude traditional fuzzers, having discovered 24 new bugs across HotSpot, ART, and R8. Among these, 20 bugs are confirmed by developers and 16 bugs hinge on intricate inter-class structures.

Keywords

Compiler Testing, Program Optimization, Inter-Class Structures

ACM Reference Format:
Shiyu Qiu, Ming Wen, Zifan Xie, and Hai Jin. 2026. Fuzzing Java Optimizing Compilers with Complex Inter-Class Structures Guided by Heterogeneous Program Graphs. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773212

*National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Huazhong University of Science and Technology (HUST), Wuhan, 430074, China
†Hubei Engineering Research Center on Big Data Security, Hubei Key Laboratory of Distributed System Security, School of Cyber Science and Engineering, HUST.
‡Corresponding author
§Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST.



ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773212

1 Introduction

Java is a widely adopted programming language renowned for its cross-platform compatibility, achieved through compilation into bytecode executed by the *Java Virtual Machine* (JVM). To enhance performance and efficiency, modern JVMs incorporate optimizing compilers such as *Just-In-Time* (JIT) [22, 25, 34] and *Ahead-Of-Time* (AOT) compilers [17, 21], which transform code while preserving program semantics. JIT compilers, like HotSpot in OpenJDK, dynamically optimize frequently executed code regions at runtime based on profiling data. In contrast, AOT compilers, such as R8 [13, 35, 57] in the Android ecosystem, perform static analysis and optimization during the build process to reduce app size, obfuscate code, and enhance security. These optimizations are required to maintain semantic equivalence to prevent functional errors or system crashes. Furthermore, *Android Runtime* (ART) adopts a hybrid approach by combining AOT compilation at installation with JIT optimization during execution, thereby achieving the balance between fast app startup and sustained runtime performance.

Object-Oriented Programming (OOP) creates software on the basis of `Objects`, which encapsulate both states and behaviors [39]. Java embraces this paradigm through the use of `Classes`, which act as blueprints for instantiating objects by defining their shared attributes and methods. Beyond the definition of individual objects, Java provides a rich set of mechanisms to structure and coordinate interactions among objects through what we term as ***inter-class structures*** in this study, such as inheritance, interface implementation, and nested classes. These structures are essential for building modular, extensible, and maintainable software systems, as they enable code reuse, abstraction, and polymorphism. For example, inheritance allows a class (the derived class) to inherit properties and behaviors from another class (the base class), simplifying the extension of functionality and reducing code duplication.

The complex relationships formed by inter-class structures are crucial for Java compilers to perform various optimizations during compilation. A flawed inference and analysis towards such relationships can lead to incorrect optimizations, causing logical errors or even program crashes. As an example, Listing 1 details a bug we discover in HotSpot's JIT optimizing compiler [49]. This issue stems from the flawed analysis in terms of a specific inter-class structure, which ultimately leads to incorrect compilation. Specifically, when Class C1 creates an instance of Class C2 (line

```
1  class C1 {
2    static String str;
3    void foo() { new C2(str); }
4  }
5  class C2 {
6    static { C1.str = "hello"; }
7    C2(String str) {
8      if (str == null)
9        throw new Exception("should not null");
10 }}
11 // call C1.foo(), expected no exception
```

Listing 1: A mis-compilation bug (i.e., JDK-8357782 [49]) in HotSpot JIT compiler caused by the flawed analysis of an inter-class structure

3), C2's static initializer block (line 6) modifies the static field `str` in C1 before its constructor is executed (line 7). The HotSpot JIT optimizer fails to correctly analyze the order of this cross-class field access, and erroneously assumes that `str` is still `null` during the constructor's execution, thus leading to a semantic error. The inter-class structures in the test case are essential to trigger this bug. According to our preliminary study (see Section 3.1), such bugs are pervasive in production compilers. For instance, for 76.4% of the historical bugs in the R8 compiler, the corresponding bug-triggering test cases contain complex inter-class structures. Therefore, it is crucial to validate Java optimization compilers to evaluate if they can optimize programs with complex inter-class structures.

Unfortunately, existing Java compiler fuzzing techniques struggle to generate test cases that can reveal bugs related to inter-class optimizations. Current approaches fall into two main categories: *generation-based* [3, 18] and *mutation-based* [14, 23, 54, 56, 58, 59, 61, 62] fuzzers. However, ***both largely focus on intra-class or intra-method structures, limiting their ability to create complex, multi-class test programs.*** For example, tools like JITFuzz [54] and MopFuzzer [56] target single-class constructs or specific JVM optimizations, while others like Jetris [61] rely on single-method code snippets. As a result, these tools are ineffective at exploring the rich inter-class interactions necessary to expose certain types of compiler optimization bugs. This highlights the *urgent need for a new fuzzing approach that can systematically generate test cases with diverse and complex inter-class structures.*

However, generating test cases with complex inter-class structures remains a significant challenge. First, modeling such structures is difficult due to the variety of mechanisms classes can interact, through instantiation, method calls, or field access, each involving diverse and intricate syntactic constructs (Challenge 1). Existing approaches largely focus on intra-class logic or support only limited inter-class interactions, making them inadequate for capturing the full complexity of class relationships. Second, generating and manipulating programs with complex inter-class structures is non-trivial, which requires careful adherence to various syntactic and semantic constraints. Otherwise, invalid test cases can be easily generated (Challenge 2). Third, incorporating inter-class structures into fuzzing seeds greatly expands the search space of test case generation, as the diverse syntactic variations lead to a combinatorial explosion. Consequently, even small changes can significantly affect semantics, making it hard to generate effective test cases efficiently without proper guidance (Challenge 3).

To address the above challenges, we present InterFuzz in this study. To address Challenge 1, we propose a novel concept of ***Heterogeneous Program Graph*** **(HPG)**, which offers a unified, high-level abstraction to represent diverse inter-class structures. Particularly, it models these structures as directed edges connecting program entities, making it easier to analyze and manipulate relationships among classes. For Challenge 2, we design a series of **Inter-Class Mutators** to create complex inter-class structures in Java code. Such mutators transform the challenging task of generating complex inter-class structures into a more tractable, high-level process of graph manipulation. For Challenge 3, we utilize a **Graph Complexity** metric based on the diversity and scale of edge types per node to guide fuzzing, which captures the variety and complexity of inter-class structures. InterFuzz prioritizes mutators likely to increase graph complexity, generating test cases with richer inter-class structures to better expose compiler optimization defects.

Extensive evaluations demonstrate InterFuzz's superior performance in uncovering compiler optimization bugs. In particular, InterFuzz has successfully identified 24 new bugs in three major Java compilers (i.e., HotSpot, Android Runtime, and R8). Among them, 20 have been confirmed by the official developers. What stands out is that 16 of these confirmed bugs depend on complex inter-class structures, which traditional fuzzers struggle to cover. Within a 24-hour testing period, it outperforms all baselines in the number of bugs detected and achieves greater code coverage on ART and R8, compilers with a strong focus on inter-class optimization. InterFuzz's high efficiency is primarily attributed to its core mechanism of graph-complexity guidance. Ablation studies reveal that, compared to a random mutation strategy, this mechanism steers test cases toward more complex structures, thereby boosting its capability to find sophisticated bugs in optimizing compilers.

To summarize, this study makes the following key contributions.

- **Originality:** To our best knowledge, we are the first to validate Java compilers in terms of the perspective of inter-class structures.
- **Approach:** We propose a concept of *heterogeneous program graph* to represent inter-class structures, based on which we design a novel approach InterFuzz to fuzzing Java compilers via generating seeds with complex inter-class structures.
- **Evaluation:** We substantiate our idea as an automated testing tool, InterFuzz, which has detected 9 bugs in R8, 4 in OpenJDK, and 7 in ART. Among these, 16 bugs are identified as inter-class related.
- **Open-Source:** Our idea is highly extensible and can be easily generalized to other languages. We thus open-source our tool InterFuzz and all the reported bugs to facilitate future research at: https://github.com/CGCL-codes/InterFuzz.

2 Background

2.1 Java Optimizing Compilers

Java code is typically compiled into bytecode by the *javac* compiler and then executed by the *Java Virtual Machine* (JVM). To enhance the performance and efficiency of Java programs, JVMs often include various optimizing compilers, such as *Just-In-Time* (JIT) [22, 25, 34] and *Ahead-Of-Time* (AOT) [17, 21] compilers, which enhance

Java code performance or reduce its size while ensuring program semantic equivalence to avoid system errors or crashes.

JIT compilers, such as HotSpot [4, 41] of OpenJDK, optimize frequently executed "hot spots" code at runtime, dynamically improving efficiency based on real-time execution conditions. For example, in the HotSpot JVM, frequently called methods within a loop are typically inlined directly into the loop body, reducing the overhead of method invocation. By comparison, **AOT compilers**, such as R8 [13, 57] in the Android ecosystem, analyze and optimize the codebase before runtime. Additionally, the *Android Runtime* (ART) [11] employs a sophisticated **hybrid strategy**. It uses AOT compilation when an app is installed to ensure fast startup times, and then leverages a built-in JIT compiler during execution to further optimize performance by profiling and recompiling hotspots on the fly. Optimizations in these compilers must strictly preserve semantic equivalence [53], as any violation will result in a defective program and critical failures such as incorrect calculations or system crashes.

In this study, we select HotSpot, ART, and R8 as our subjects because these optimizing compilers are widely used. In 2024, over 90% of JDK distributions use HotSpot or a HotSpot-based JVM [20], 71.85% of mobile phones use the Android operating system, and 1.68 million apps are available in the Google Play Store [9]. Meanwhile, R8 is the default compiler in the Android build process.

2.2 Inter-Class Structures in Java

Object-Oriented Programming (OOP) creates software on the basis of `Object`. Java implements this paradigm through `Class` and `Interface`, constructs that act as the blueprint for creating objects by defining their common structures and behaviors. Additionally, Java offers diverse mechanisms for handling interactions among objects through what we term ***inter-class structures***, such as inheritance, interface implementation, and nested classes. For instance, inheritance is a mechanism in OOP that enables a class, known as the derived class, to inherit properties and methods from another class, referred to as the base class.

The Java Language Specification introduces complete inter-class structures. Specifically, take the Java 17 Language Specification [32] as an example, which is the most widely used version now [38], it declares a `Class` as follows: `ClassDeclaration: {ClassModifier} class TypeIdentifier [TypeParameters] [ClassExtends] [Class Implements] [ClassPermits] ClassBody`, through which we can establish various relationships among classes:

- `TypeParameters` constrain the types a generic class can handle, forming **Generic Bounds** as an inter-class structure.
- `ClassExtends` establishes an **Inheritance** structure.
- `ClassImplements` declares the interfaces a class implements, forming an **Interface Implementation** structure.
- `ClassPermits` restricts inheritance by specifying which classes are permitted to extend the current class.
- `ClassBody` primarily consists of fields and methods. These members can refer to other classes, such as by accessing their fields or invoking their methods. We define the reference to other classes in `ClassBody` as a **Reference** structure.

Furthermore, new classes or interfaces can be defined within the `ClassBody`, creating a **Nesting** structure. Based on the above and the declaration of interfaces, we can categorize inter-class relationships into five main types: *Inheritance*, *Interface Implementation*, *Nesting*, *Generic Bounds*, and *Reference*.

```
// Run a worklist algorithm to search for a subclass
// of the target group (including the classes in the
// target group) where adding the interfaces from
// the source group would change method resolution.
Map<DexProgramClass, DexMethodSignatureSet>
      rootDefaultInterfaceMethodsOfInterest =
 MapUtils.transform(
  toGroup,IdentityHashMap::new,Function.identity(),
  ignoreArgument(() -> defaultIntfMethods));
```

Listing 2: R8's source code that handles both *Inheritance* and *Interface Implementation* during optimizaion

The real example in Listing 1 indicates that it is crucial for compilers to correctly handle inter-class relationships when performing optimization to ensure the stability and correctness of compilers. Listing 2 shows how the R8 compiler carefully considers inter-class structures such as *Inheritance* and *Interface Implementation* when optimizing program sizes [36]. Specifically, before reducing a class, R8 verifies whether its subclasses and implemented interfaces remain compatible with the reduction. Such checks allow R8 to correctly identify and process only those classes that are genuinely safe to reduce. However, such logic is often complex, and thus prone to be defective, eventually leading to pervasive bugs (see Section 3.1 for statistics). Unfortunately, such bugs can hardly be detected by existing state-of-the-art approaches [14, 23, 54, 56, 58, 59, 61, 62] since all of them have overlooked the inter-class structures of seeds when performing fuzzing.

3 Motivation and Insights

3.1 Pervasiveness of Inter-Class Bugs

To further highlight the importance of testing Java optimizing compilers from the inter-class perspective, we first investigate the pervasiveness of bugs triggered by inter-class structures. In particular, we perform a preliminary study based on all historical compiler bugs to investigate whether their bug-revealing programs include inter-class structures. Specifically, we focus on widely used Java compilers (i.e., HotSpot, Android Runtime (ART), and R8), and collect historical bug-revealing programs from their mainline repositories. Following previous work [62], we collect all bug-revealing programs from the test suites associated with historical bugs of the above compilers. In total, we collect 4,127 programs for HotSpot, 3,547 for R8, and 1,075 for Android Runtime, respectively.

We leverage Spoon [33] to analyze the code structures of the corresponding bug-revealing programs. Specifically, we identify **Inheritance** and **Interface Implementation** structures by matching the `extends` and `implements` keywords. **Nesting** structures are detected by checking whether a class or interface is defined inside another class or interface. We recognize **Generic-Bound** structures by checking for bounded type parameters. **Reference** structures are identified by scanning a class's fields, methods, and constructors for any access, invocation, or instantiation involving

other classes or interfaces. A program is classified as having inter-class structures if it exhibits at least one of the five aforementioned types. A given program can contain multiple inter-class structures.

Table 1: Proportion of bugs with inter-class structures in historical bug-revealing programs and the average number of inter-class structures per test program

Compiler	#Test	#Inter-Class Test	#Avg Inter-Class Structures Per Test
HotSpot	4,127	1,111 (26.9%)	28.5
ART	1,075	536 (49.9%)	49.4
R8	3,547	2,711 (76.4%)	69.2

As indicated in Table 1, a significant portion of historical bug-revealing programs in Java compilers rely on inter-class structures. In HotSpot, 1,111 (26.9%) of these programs contain such structures. This proportion is even higher in R8 and ART, at 2,711 (76.4%) and 536 (49.9%), respectively. This portion may be over-approximated since we do not verify whether the bugs can only be triggered with inter-class structures. However, the impact is limited as our inspection procedure follows standard practice in compiler development and bug fixing. In particular, when developers fix a bug, they first reduce the bug-triggering program to a *Minimal Reproducible Example* (MRE) [37, 43] to isolate the root cause. For example, HotSpot records JDK-8290910 [46] and JDK-8260370 [47], where developers use C-Reduce [8], a test-case reduction tool, to obtain the MRE. If the reduced MRE still contains inter-class structures (e.g., *Inheritance*), this provides strong evidence that such structures are necessary to trigger the bug. R8 is highly sensitive to inter-class structures due to its aggressive, whole-program restructuring aimed at reducing application size. Its irreversible transformations, such as redundant class elimination, depend entirely on the correctness of static analysis, making it particularly vulnerable to complex inter-class structures. In contrast, HotSpot's JIT compiler is less affected because it leverages runtime information to guide optimizations and performs less inter-class analysis. Furthermore, its *deoptimization* mechanism allows it to safely reverse speculative optimizations if runtime assumptions change. ART's hybrid approach of JIT and AOT places it between these two extremes. While its AOT optimizations are static and irreversible like R8's, they are less aggressive and more focused on performance, resulting in an intermediate level of vulnerability to bugs arising from inter-class structures.

Moreover, the nature of these necessary inter-class structures is far from simple. The average number of inter-class structures per test program, 28.5 in HotSpot, 49.4 in R8, and a striking 69.2 in ART, strongly suggests that triggering these bugs often requires a complex context woven from substantial inter-class structures. Since historical bug-revealing programs are minimized to the smallest reproducible examples, any remaining inter-class structures are, by practice, a necessary condition for triggering these bugs.

Finding: Triggering a large portion of real-world bugs in Java compilers demands more than simple code constructs. It requires test cases embodying **complex inter-class structures**.

3.2 A Motivating Example and Insights

Listing 3 illustrates a bug we discover in the *Android code shrinker* component of the R8 Compiler. The bug manifests as a mis-compilation error. In particular, instead of the correct output "I1.foo I2.foo", the program incorrectly produces "I2.foo I2.foo". Such a mis-compilation can lead to severe runtime errors, especially for large-scale systems.

```
interface I1 {
  default void foo() { System.out.println("I1.foo"); }
}
interface I2 extends I1 { //  Inheritance
  default void foo() { System.out.println("I2.foo"); }
}
class C1 implements I1 {} // Interface Implementation
class C2 implements I2 {} // Interface Implementation
class C0 {
  public static void main(String[] args) {
    C1 c1=new C1(); C2 c2=new C2(); // Reference
    c1.foo(); c2.foo(); // Reference
}}
```

Listing 3: A test case triggering a mis-compilation bug (i.e., IssueTracker-420228751 [50]) in R8. Its mutation process adds inter-class structures and inter-class entities.

The complex inter-class structures in the test case are essential to trigger the bug. In the code as shown in Listing 3, two distinct interfaces, `I1` and `I2`, are involved, where `I2` extends `I1` and both of them define a method `foo` printing the signatures. Correspondingly, classes `C1` and `C2` implement `I1` and `I2` respectively. In this example, the complex inter-class structures include *Inheritance* between `I1` and `I2`, *Interface Implementation*, and *Reference* from `C0` to `C1` and `C2`. This bug stems from a flaw in R8's class-merging process, an optimization designed to reduce program size [13]. In this case, R8 incorrectly deems `C1` and `C2` to be semantically equivalent, assuming their respective `foo()` methods are interchangeable. This misjudgment stems from its flawed analysis of the *Inheritance* between their interfaces, `I1` and `I2`. Consequently, a method call to `c1.foo`, which should resolve to `I1`'s implementation (printing "I1.foo"), is wrongly dispatched to the behavior defined in `I2` (printing "I2.foo"). Such inter-class structures are essential to trigger the bug, and removing any one of the structures, such as the *Interface Inheritance* between `I1` and `I2` or the *Inter-Class Method Invocation* between `C0.main` and `C1.foo`, prevents the mis-compilation bug, as the incorrect merge process will not be triggered.

Unfortunately, it is hard for existing Java compiler testing techniques to generate a test case that can trigger such bugs. Existing fuzzing techniques for Java compilers can be broadly classified into two categories [29, 63]: *generation-based* and *mutation-based*. However, both categories tend to focus on structures within a single class or method. As a result, they struggle to effectively generate test cases with complex inter-class structures. **Generation-based** fuzzers [3, 18] create test cases from scratch according to a predefined input model, such as a grammar. JavaFuzzer [18] is a prominent example, capable of generating test cases with numerous arithmetic operations. However, its input model does not account for inter-class structures, limiting it to generating only single-class tests. **Mutation-based** fuzzers [14, 23, 54, 56, 58, 59, 61, 62], on the

other hand, modify existing valid test cases (also known as seeds) to create new ones. While they have achieved notable success, their mutation strategies are also primarily confined to intra-class or intra-method structures. For instance, JITFuzz [54] employs four carefully designed mutators to trigger specific JVM optimizations, such as function inlining, simplification, scalar replacement, and escape analysis. However, the optimizations targeted by these mutators are intra-class, meaning the mutator does not alter the program's inter-class structures and cannot generate test cases with complex inter-class structures. Similarly, Jetris [61] generates test cases by extracting code snippets from historical bug-revealing programs. These snippets, however, typically involve only single-method logic. Consequently, the synthesized test cases often lack complex inter-class structures required to reveal certain types of bugs. In summary, such deficiencies of existing fuzzing tools prevent them from adequately uncovering bugs related to inter-class optimizations in Java compilers. Therefore, there is a clear and pressing need for a novel fuzzer capable of effectively generating tests with rich inter-class structures.

However, generating test cases with complex inter-class structures that can trigger compiler bugs is challenging. *First, modeling complex inter-class structures is challenging* (**Challenge 1**). The relationships among classes are inherently intricate, as classes can interact through various mechanisms such as instantiation, method invocation, or field access. Generating test cases that capture such inter-class interactions requires a comprehensive understanding of the language's diverse syntactic constructs. Second, *generating and manipulating test cases with inter-class structures is non-trivial* (**Challenge 2**). Generating inter-class structures necessitates careful adherence to various syntactic and semantic constraints, including access modifiers, inheritance hierarchies, and method signatures. Failure to respect these constraints may result in syntactically invalid or semantically incorrect code, ultimately leading to invalid or ineffective test cases. *Third, incorporating inter-class structures into seed generation substantially expands the search space, potentially compromising fuzzing efficiency* (**Challenge 3**). The diverse syntactic options for inter-class structures lead to an explosion of possible combinations when mutating testing seeds. Even minor changes within a combination can significantly alter the program's semantics. Therefore, without effective techniques to systematically explore this enlarged search space, the testing efficiency can be compromised.

Our idea. We address the above challenges based on the following insights. First, to tackle Challenge 1, we introduce a novel concept of ***Heterogeneous Program Graph*** **(HPG)** as a unified yet succinct representation for various inter-class structures. Specifically, we abstract the diverse syntactic implementations of inter-class structures into distinct edge types in the graph. By analyzing and manipulating graphs, we can uniformly analyze a wide variety of inter-class structures. Second, for Challenge 2, we design **Inter-Class Mutators** that generate complex inter-class structures within a seed program via manipulating the HPG of the program. By abstracting concepts like inheritance or reference into distinct edge types, we transform the challenging task of mutating inter-class structures into a more tractable, high-level process of graph manipulations. Third, for Challenge 3, we introduce a **Graph Complexity Metric** for HPGs that captures two key aspects of inter-class structures: the diversity of structure types and their total quantity. A higher complexity indicates a greater variety and number of edge types, reflecting richer and more numerous inter-class structures in the corresponding program. The fuzzing process is then directed to prioritize mutators that increase HPG's complexity, enabling it to efficiently navigate the search space to generate complex test cases that are more likely to trigger compiler bugs.

4 Approach

We substantiate the above idea as an automated fuzzing tool, InterFuzz, and Figure 1 shows its overview. InterFuzz begins with a Java seed file. In **Step 1**, this file is parsed to construct an initial HPG. In **Step 2**, a mutator is chosen from a pool of inter-class mutators, which is designed to add inter-class structures by expanding the HPG, through weighted random selection. This mutator is then applied to the current graph and Java file, producing a new, more complex HPG and an updated Java file. Then, the resulting change in graph complexity is computed and used for **Step 3**, where the weight of the chosen mutator is updated to guide the selection of the most effective mutator in the future. After a set number of iterations, InterFuzz generates a complex HPG and the corresponding Java test case file with complex inter-class structures. This generated file is then used to test the target Java optimizing compilers (e.g., OpenJDK or ART), specifically to validate their analysis and optimizations of complex inter-class structures through differential testing.

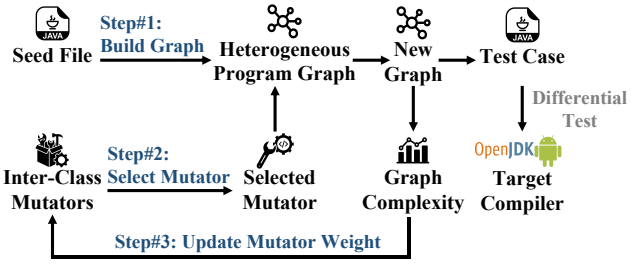


Figure 1: Overview of InterFuzz

4.1 Heterogeneous Program Graph

The variety of inter-class structures in Java, such as *Inheritance*, *Interface Implementation*, and *Reference*, poses a significant challenge for unified analysis and manipulation. For example, a reference structure between two classes can be established through class instantiation, method calls, or field access. Similarly, interface implementation is not limited to the implements keyword. It can also be achieved using *lambda expressions* for functional interfaces or anonymous classes for temporary implementations. To tackle the challenge of representing these structures uniformly, we introduce the concept of ***Heterogeneous Program Graph*** **(HPG)** to represent such structures. HPG is a *directed graph* and contains multiple types of nodes and edges, making it ideal for modeling complex systems with various entities and connections. The participating entities, such as classes and methods, are treated as nodes. The structural relationships between them, such as extends, implements, or calls, are represented as distinct types of edges.

Definition 4.1 (*Heterogeneous Program Graph* (HPG)). Following existing studies [55, 60], HPG for a program $\mathcal{P}$ is defined as $HPG(\mathcal{P})$, $HPG(\mathcal{P}) = (V_{\mathcal{P}}, \mathcal{T}, E_{\mathcal{P}}, \mathcal{E})$. Here, $V_{\mathcal{P}}$ is the set of inter-class entities of $\mathcal{P}$, as the nodes of the graph, with the set of types defined as $\mathcal{T}$. Similarly, $E_{\mathcal{P}}$ is the set of inter-class structures of $\mathcal{P}$, as the edges of the graph, with the set of types defined as $\mathcal{E}$.

The HPG design offers a high-level abstraction of Java's inter-class structures by unifying diverse syntactical details into a graph representation. For instance, a single edge type, such as *Reference*, can represent a wide range of concrete implementations, including inter-class method calls and field accesses. Similarly, the HPG design simplifies and standardizes the manipulation of inter-class structures. For example, we can generate or adjust these structures by simply adding or modifying nodes and edges on the HPG. Furthermore, the HPG allows us to reframe the search for complex class combinations as a problem of graph expansion. In essence, this transforms the abstract and vague concept of "structural complexity" into a concrete, analyzable mathematical model as "graph complexity". Such a design enables InterFuzz to quantify the complexity of inter-class structures and leverage this metric to guide the generation of more intricate test cases.

Definition 4.2 (Inter-Class Entity (Node)). In an HPG, nodes represent the program's inter-class entities, which are the fundamental components involved in inter-class structures. To concisely cover inter-class structures, we identify four types of entities, with the set of node types denoted as $\mathcal{T}$:

$$\mathcal{T} = \{Class, Intf, Mtd, Fld\} \tag{1}$$

Class (*Class*) and interface (*Intf*) are the core components in inter-class structures such as *Inheritance*, *Interface Implementation*, and *Nesting*, while method (*Mtd*) and field (*Fld*) participate in the *Reference* structures. For a program $\mathcal{P}$, the node set $V_{\mathcal{P}}$ of its HPG represents all inter-class entities in the program. Each node $v \in V_{\mathcal{P}}$ is a tuple $v = (t, n)$, where t is the node type, belonging to the node type set $\mathcal{T}$ (i.e., $t \in \mathcal{T}$), and n is the node attribute, which refers to the name of the entity or other relevant information (e.g., for the method `I1.foo` in the motivating example, the node type t is *Mtd*, and n includes details like it is a non-static method).

The design of nodes aims to cover multiple similar Java entities in a unified way. A *Class* node, for instance, can represent not only a regular class but also an enum or a record. To distinguish between them, we use the node's attributes (i.e., n) to record specific features.

Definition 4.3 (Inter-class Structures (Edge)). For a program $\mathcal{P}$, the edge set $E_{\mathcal{P}}$ represents the collection of all inter-class structures in the program. The edges in the HPG are directed edges. Each edge $e \in E_{\mathcal{P}}$ is a tuple $e = (v_s, v_t, r, n)$, where v_s is the source node, v_t is the target node, and r is the edge type (i.e., $r \in \mathcal{E}$), and n is the edge attribute capturing additional properties of the related inter-class structure. The HPG contains five edge types, with the set of edge types denoted as $\mathcal{E}$.

$$\mathcal{E} = \{\ Inheritance, Interface\ Implementation, Nesting, Generic\ Bounds, Reference\} \tag{2}$$

As previously stated, Java offers a rich variety of constructs for implementing inter-class structures. However, we limit our analysis to the most common syntactic forms in the current design as shown in Table 2 for the following reasons. First, programming languages are inherently dynamic, with new features being introduced continuously, making it challenging to model all possible constructs comprehensively. Moreover, the primary goal of our work is to validate the effectiveness of the proposed inter-class fuzzing approach. Second, our evaluation shows that even when limited to five common structural patterns, the approach remains effective in uncovering compiler bugs (see Section 5). Finally, our approach is highly extensible; it can be easily adapted by incorporating additional node or edge types. Furthermore, the underlying idea is language-agnostic and can be applied to other programming languages as well.

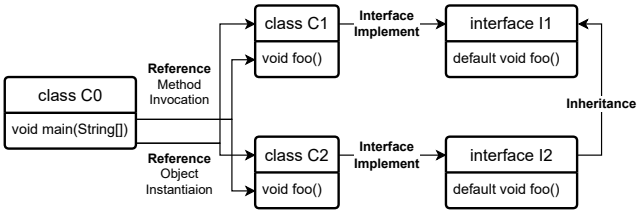


Figure 2: HPG of the motivating example

The HPG models program semantics at the granularity of inter-class structures. For instance, Figure 2 shows the HPG of the motivating example in Section 3.2. Specifically, a *Reference* edge from method `C0.main` to classes `C1` and `C2` signifies the instantiation of these classes within that method. The *Inheritance* edge from interface `I2` to `I1` indicates that `I2` extends `I1`.

4.2 Inter-Class Graph Mutators

Manipulating Java test code with complex inter-class structures is challenging. Directly modifying source code to add these structures is not only complex but also risks violating Java's syntax constraints. By utilizing the *Heterogeneous Program Graph* (HPG) defined in Section 4.1, we can transform the challenging task of generating Java programs with complex inter-class structures into the manipulation of nodes and edges on the HPG. In particular, InterFuzz designs a set of **Inter-Class Mutators** that proactively mutate inter-class structures of a program by operating on the corresponding HPG.

A core principle for inter-class mutators is modularity: each mutator should perform an atomic operation on the HPG. This design minimizes interference between mutators, reducing the disruption caused by adding new structures to existing ones. It also enables the construction of complex inter-class structures by combining simple operations, which is exactly the goal of our generation process. Specifically, we categorize inter-class mutators into two types: *node addition* and *edge addition*. **Node addition** involves adding a new node to the graph, which corresponds to creating a new inter-class entity in the program. For example, InterFuzz can apply an "Add Class Node" mutator to insert a class node in the graph, generating a new class in the code. **Edge addition**, on the other hand, involves adding an edge between existing nodes, which introduces a new inter-class structure. For instance, applying an "Add Inheritance Edge" mutator connects two class nodes in the graph, making one class inherit from another in the program. These atomic operations

Table 2: Edge type definition in the heterogeneous program graph. Edges are directed, from "Source Node (A)" to "Target Node (B)", with "Description" indicating the inter-class structure represented by the edge and "Example" providing the code text.

Inter-Class Structure	Source Node	Target Node	Description	Example
Inheritance	Class A	Class B	A inherit B with extends keyword	class A extends B
	Intf A	Intf		interface A extends B
Interface Implementation	Class A	Intf	A implements B with implements keyword	class A implement B
Nesting	Class/Intf A	Class/Intf B	Static class/interface B or is defined inside class/interface A.	class A { static class B { ...} }
	Class/Intf A	Class B	Non-static Class B is defined inside class/interface A.	class A { class B { ...} }
Generic Bounds	Class/Intf A	Class/Intf B	Constrains the type parameter T of A to be a subtype of B.	class A⟨T extends B⟩
Reference	Mtd A	Class/Intf B	Method A instantiate class/interface B, which not define A.	void A { B b = new B(); }
	Mtd A	Fld B	Method A accesses field B, which is located in another class.	void A { someClass.B; }
	Mtd A	Mtd B	Method A invokes method B, which is located in another class.	void A { B b = new B(); }

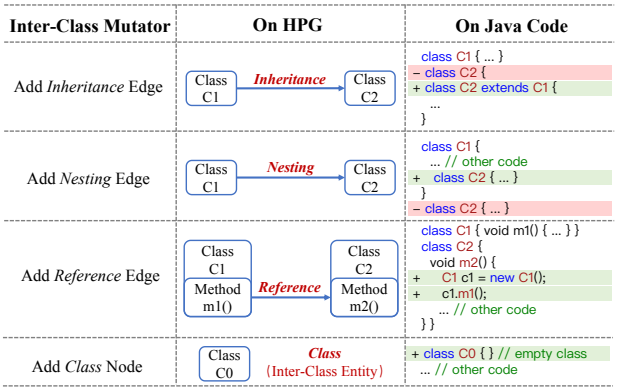


Figure 3: Three inter-class mutators (4/9) designed in InterFuzz to generate inter-class structures

reduce dependencies between changes, making it more tractable and flexible to introduce complex inter-class structures. Notably, InterFuzz does not need to convert the HPG back to Java code. Instead, it constructs the corresponding HPG for each seed program, using it solely as an abstract representation serving for the mutation and guiding process.

In practice, we design mutators for five types of inter-class structures (i.e., five edge types in HPG) and four types of inter-class entities (i.e., four node types in HPG). Each mutator performs node or edge addition on the HPG, which modifies the corresponding class relationships in the Java test case code. These mutators are designed to prevent syntactically invalid or semantically incorrect code. By design, they account for potential issues such as cyclic calls, cyclic inheritance, and multiple inheritance, which are carefully handled within InterFuzz's implementation. For instance, when adding an *Inheritance* edge, InterFuzz verifies that the new edge does not introduce an inheritance cycle in the HPG, in compliance with the Java language specification. Due to page limit, we only describe four representative mutators in detail as shown in Figure 3, with others accessible in our artifact website.

Add *Inheritance* edge. This mutator introduces *Inheritance* structures into the program. InterFuzz selects two compatible nodes (e.g., classes `C1` and `C2`) and connects them with an *Inheritance* edge, which translates to an *extends* declaration in the source code.

Add *Nesting* edge. This mutator creates *Nesting* structures. It models a refactoring on the HPG by adding a directed nesting edge from an outer class to an inner class. For instance, adding an edge from `C1` to `C2` corresponds to transforming `C2` into a nested type within the scope of `C1`.

Add *Reference* edge. This mutator introduces reference structures between classes in the seed program. These structures can take various forms, such as class instantiations, method calls, and field accesses. For example, a reference structure can be established by calling a method from one class within another method. As shown in Figure 3, class `C1` defines method `m1`, and class `C2` defines method `m2`. A reference relationship can be established in method `m1` by instantiating class `C2` and calling its method `m2`. In the HPG, this corresponds to a reference edge from method node `m1` to method node `m2`.

Add *Class* node. This mutator adds a new class node (e.g., `C0` in Figure 3) to the HPG, creating a new class. Although this does not directly introduce inter-class structures, subsequent mutations can apply other mutators to the new class `C0`.

The inter-class mutators used in InterFuzz are designed to be extensible. If more inter-class structures need to be considered, future work can expand the types of nodes and edges in the HPG to represent additional inter-class structures or capture more details about them. At the same time, the design of inter-class mutators can be extended to align with these structures. Moreover, the HPG and inter-class mutator design are language-agnostic. Future work can adapt them for other OOP languages, such as C++.

4.3 Graph Complexity Guidance

As aforementioned, test cases with complex inter-class structures are vital for validating Java optimizing compilers. During the mutation process, our goal is to generate seeds with intricate inter-class structures, and our insight is simple and adheres to the empirical statistics as shown in Table 1: *test cases with more complex inter-class structures are more likely to trigger diverse behaviors related to inter-class optimizations, thus uncovering potential bugs.* Therefore, we quantify a graph's complexity from two aspects: *edge entropy* and *node degree*. The edge entropy measures the diversity of edge types associated with a node, while the node degree measures the number of edges connected to a node. We comprehensively consider the edge entropy of each node's connected edges and the degree of each node as its complexity, and then sum the complexities of all nodes to obtain the complexity of a graph. A higher graph complexity, therefore, indicates that a program contains richer inter-class structures. This graph complexity guides the mutation process of InterFuzz. InterFuzz selects a mutator based on its assigned weight

(set equally initially). After applying a mutation, InterFuzz recalculates these weights based on changes in terms of graph complexity. This rewards mutators that are more likely to increase complexity, boosting their weights for future selections.

The **edge entropy** of a node e measures the diversity of edge types associated with that node [40], defined as

$$H(e) = -\sum_{t \in \mathcal{E}} p(t) \log p(t) / \log |\mathcal{E}| \tag{3}$$

where $\mathcal{E}$ is the set of edge types, and $p(t)$ is the proportion of edges of type t among the edges connected to node e. If a node has multiple types of edges (e.g., a class that inherits from another class and implements multiple interfaces), its edge entropy is high.

Similarly, the **node degree** of a node e, denoted as $\mathrm{Deg}(e)$, measures the number of edges connected to the node [2]. A high node degree indicates that the node is connected to more edges, participating in a larger number of inter-class structures. For example, if a method node is called by multiple inter-class functions or calls multiple inter-class functions, its degree will be high.

In HPG, we define the complexity of a node as the product of its edge entropy and degree marked as $C(e)$, and a graph's complexity is the sum of the complexities of all nodes as

$$C(HPG(\mathcal{P})) = \sum_{e \in V_{\mathcal{P}}} C(e) = \sum_{e \in V_{\mathcal{P}}} H(e) \cdot \mathrm{Deg}(e) \tag{4}$$

The above definition offers two merits. (1) Node complexity integrates both the diversity of edge types and the number of edges, providing a comprehensive measure of a node's involvement in inter-class structures. (2) Graph complexity reflects the overall diversity of inter-class structures in the program, facilitating guidance for mutation operations.

To guide the mutation process, we define Δ to measure the complexity change of an HPG before and after applying a mutator:

$$\Delta = C(HPG(\mathcal{P}')) - C(HPG(\mathcal{P})) \tag{5}$$

where $HPG(\mathcal{P})$ represents the graph before mutation, and $HPG(\mathcal{P}')$ represents the graph after mutation. A positive Δ indicates that the mutator has successfully increased the graph's complexity, while a negative or zero Δ suggests it has not. We assign each mutator an initial weight, which determines its probability of being selected. The probability of selecting a mutator m is:

$$P(m) = weight(m) / \sum_{m' \in M} weight(m') \tag{6}$$

where $weight(m)$ is the weight of graph mutator m, and M is the set of all graph mutators. After each mutation, the weight of the applied mutator is updated based on Δ:

$$weight(m) = weight(m) + \Delta \tag{7}$$

If $\Delta > 0$, the mutator's weight increases, favoring its selection in future iterations. If $\Delta \leq 0$, the weight decreases, reducing its likelihood of selection.

Design principles of the weighting mechanism. By using node complexity as the key metric, we assess the variety and number of inter-class structures, rewarding mutators that create diverse inter-class patterns. We avoid simplistic metrics like node or edge counts since they tend to produce ineffective graphs as they ignore the connections among nodes or edges. On the contrary, our complexity formula $C(e) = H(e) \cdot \mathrm{Deg}(e)$ emphasizes the connections among nodes and edges, as well as their diversity, thus creating more effective graphs. For example, by introducing diverse structures over time instead of repeating the same one, we achieve two effects. This approach increases the node's degree, $Deg(e)$, by adding new connections, and also raises its edge entropy, $H(e)$, by adding variety. The combined growth boosts the node's complexity, which aligns with our goal of generating complex inter-class structures. Consequently, the weight mechanism steers mutations toward complex HPG and inter-class structures in the program.

4.4 Test Oracle and Test Case Reduction

InterFuzz employs the following two test oracles to detect bugs:

Crash. A bug is detected if the JVM crashes during execution. The cause is analyzed using the corresponding crash log, such as `hs_err_pid.log` for HotSpot JVM. Other JVM implementations, like R8 and ART, generate their own crash logs for similar analysis.

Semantic error. InterFuzz compares the outputs of different Java compilers to detect semantic errors. The process is as follow:

(1) InterFuzz compiles the generated test case with `javac` to get JVM bytecode.
(2) It runs R8 to translate the JVM bytecode into ART bytecode.
(3) It executes the JVM bytecode on JVM and records the output O_{JVM}, and executes the ART bytecode on ART to obtain O_{ART}.
(4) A semantic error is detected when O_{JVM} and O_{ART} differ.

Such inconsistencies may come from any of the three compilers: R8, the JVM, or ART. To isolate the root cause, we manually inspect each case and determine which compiler is buggy.

Test case reduction. InterFuzz generates test cases with complex inter-class structures, and thus the resulting programs tend to be large and difficult for developers to understand. In practice, many bugs can still be triggered by only a few key structures. We therefore adopt a semi-automated approach to reduce test cases. We first apply a state-of-the-art test case reduction tool [8, 44] and then manually remove non-essential inter-class structures until further deletion no longer triggers the bug.

5 Evaluation

This section presents our experimental design for evaluating the usefulness and effectiveness of InterFuzz. The evaluation is guided by answering the following research questions:

- **RQ1:** How effective is InterFuzz in detecting bugs?
- **RQ2:** Does InterFuzz outperform other Java compiler testing approaches?
- **RQ3:** What is the impact of InterFuzz's key design components?

5.1 Experiment Setup

Target Java optimizing compilers. We choose three widely used Java optimizing compilers as our targets: HotSpot, Android Runtime (ART), and R8. For HotSpot, we select OpenJDK [7], a highly popular implementation of the HotSpot virtual machine. Specifically, we build the mainline development branch for JDK 26 at the commit hash `ee0d30` [31]. For ART, we target the latest version of the *Android Open Source Project* (AOSP) [1]. We build ART from the AOSP mainline repository at the commit hash `648461`. For R8, we

compile the compiler from the mainline branch of Google's official R8 repository [16], using the commit hash `922e1a`.

Environment. We run our experiments on a Linux server with dual Intel Xeon Gold 6248R CPUs and 256GB of RAM. We implement the tool's design using *Spoon* [33], an open-source library for analyzing, rewriting, transforming, and transpiling Java source code. To measure code coverage, we compile the Java optimizing compiler with *gcov* [10, 26] and use *JaCoCo* [12] to collect and analyze coverage data. For HotSpot, we use its built-in `-enable-native-coverage` option to enable *gcov* support. For ART, we enable *gcov* support with the built-in `NATIVE_COVERAGE=true` option. For R8, we use *JaCoCo* for runtime instrumentation to gather coverage information. Both *gcov* and *JaCoCo* are widely used tools that provide detailed code coverage information.

Parameters. Our practice suggests that 200 iterations can balance generation efficiency and effectiveness when exploring inter-class structures. As InterFuzz introduces complex inter-class structures, an excessively complex seed will degrade compilation and execution efficiency. Therefore, we set the maximum number of iterations to 200. We also initialize the weight of each inter-class mutation operator to 0.1 at the start of fuzzing iterations.

Seed pools. Following prior work, we use *JavaFuzzer* [18] to generate initial seed test cases. *JavaFuzzer* can generate numerous small, random test cases using predefined heuristics to thoroughly test the Java VM compiler and runtime across diverse scenarios. The seeds generated by *JavaFuzzer* contain naive inter-class structures. Each test case only consists of two classes: *Test* and *FuzzerUtils*. The only inter-class interaction between the *Test* class and *FuzzerUtils* involves basic references through method calls, without inheritance, interfaces, or other complex dependencies.

Baselines. InterFuzz is the first to propose a testing mechanism for analyzing and optimizing inter-class structures in Java optimizing compilers. To demonstrate its effectiveness in detecting issues with inter-class structures in Java optimizing compilers, we compare it with four state-of-the-art baselines: Artemis [23], MopFuzzer [56], JITFuzz [54], and Jetris [61]. Artemis, MopFuzzer, and JITFuzz are designed for JVM JIT testing, primarily focusing on mutating intra-class or even intra-method structures to evaluate JIT optimization behaviors. Jetris tests the JVM as a whole by generalizing bug-revealing patterns from historical bugs and using these patterns to guide the generation of additional program components. While existing work does not focus on testing inter-class structures, their generated test cases might still contain various structures, potentially triggering certain bugs. For a fair comparison, we use the same pool generated by *JavaFuzzer* for all baseline works.

5.2 RQ1: Bug Detection Ability of InterFuzz

Detected bugs. We conduct a three-month test on the latest versions of HotSpot, ART, and R8 using our tool. InterFuzz detects 24 bugs in total, with 20 confirmed as genuine by developers (4 for HotSpot, 7 for ART, and 9 for R8). Detailed bug statistics are shown in Table 3. Bugs in the *Confirmed* state are issues that developers have acknowledged but have not yet fixed. The high number of bugs found in R8 stems from its primary goal, which focuses on whole-program optimization, aiming to reduce the app size. This process requires analyzing and aggressively optimizing extensive inter-class structures. Consequently, the complexity of its implementation leads to a higher potential for hidden bugs. On the other hand, InterFuzz detects fewer bugs for HotSpot since it focuses on improving runtime performance by optimizing the execution of "hot" code snippets. Therefore, it performs fewer complex analyses and optimizations across classes, and thus contains fewer bugs.

Table 3: Breakdown of detected bugs by InterFuzz

No.	Bug ID	Compiler	Symptom	Status	Inter-Class Related	Priority
1	8357782	HotSpot	Semantic Error	Fixed	✓	P3
2	8361699	HotSpot	Crash	Fixed	✓	P3
3	8357381	HotSpot	Crash	Confirmed	✓	P3
4	8357242	HotSpot	Semantic Error	Duplicate	✗	P4
5	405152615	ART	Semantic Error	Fixed	✗	P2
6	418897611	ART	Semantic Error	Confirmed	✓	P2
7	369670481	ART	Semantic Error	Confirmed	✓	P3
8	369739225	ART	Semantic Error	Confirmed	✓	P3
9	405149431	ART	Semantic Error	Confirmed	✗	P3
10	410253904	ART	Semantic Error	Confirmed	✗	P3
11	405149432	ART	Semantic Error	Duplicate	✓	P3
12	412524379	R8	Crash	Fixed	✓	P1
13	419464490	R8	Crash	Fixed	✓	P1
14	379347946	R8	Semantic Error	Fixed	✓	P1
15	420228751	R8	Semantic Error	Fixed	✓	P2
16	367915233	R8	Semantic Error	Fixed	✓	P2
17	426351560	R8	Semantic Error	Fixed	✓	P2
18	418719343	R8	Semantic Error	Fixed	✓	P2
19	419404081	R8	Semantic Error	Confirmed	✓	P2
20	372806451	R8	Semantic Error	Duplicate	✓	P2

Among the bugs detected by InterFuzz, 4 directly cause compiler crashes, while 16 result in semantic errors in the compiled output. Semantic errors refer to cases where the program optimized by the compiler produces unexpected behavior or runtime errors. For compiler testing, detecting semantic errors is crucial, as they can lead to unpredictable program behaviors or crashes during execution. However, semantic errors are often hard to uncover because they typically stem from logical flaws in the compiler's analysis and optimization. The presence of these bugs shows that InterFuzz can effectively probe subtle logical errors in optimizing compilers.

Inter-class related bugs. Among the 20 confirmed bugs, 16 require inter-class structures to trigger: 3 in HotSpot, 4 in ART, and 9 in R8. To simplify each bug test case while preserving the bug, we use reduction tools Perses [44] and CReduce [8], combined with manual reduction. For the resulting reduced test cases, we apply the same criteria as in the preliminary study (Section 3.1) to assess whether inter-class structures are necessary to trigger the bugs. If a bug's reduced test case involves any of these structures, we consider it requires inter-class structures to be triggered. These 17 inter-class related bugs cannot be detected by other baseline tools. The inter-class structures they involve are absent in *JavaFuzzer* seeds, and the mutation processes of other baselines cannot introduce these complex inter-class structures required to trigger the bug.

These inter-class triggered bugs are a long-standing challenge for Java compilers and directly affect developers. When the compiler crashes, it can delay app releases. A representative case is JDK-7064302 [45], where a complex inter-class structure triggers a HotSpot optimization error that causes Tomcat to crash at runtime. Furthermore, the inter-class related bugs we find are considered high-priority by developers. The HotSpot team labels JDK-8361699 [48] as "Impact=High" and fixes it promptly. The R8 team

fixes our reported issue (Issue 412524379 [52]) within two hours, which highlights the practical importance of these findings.

```
1  class C0 {
2    interface I1 {
3      default void foo() {
4        System.out.println("C0.I1.foo");
5    } }
6    interface I2 {
7      default void foo() {
8        System.out.println("C0.I2.foo");
9    } }
10   class C1 implements I1, I2 {
11     public void foo() {
12       I1.super.foo();
13 } } }
```

Listing 4: An R8 compiler crash bug (i.e., Issue-8357782 [51]) caused by the flawed analysis of *Nesting* structure

Here, we use a bug detected by InterFuzz to illustrate how it validates the handling of inter-class structures in optimizing compilers. Listing 4 shows a simplified test case generated by InterFuzz that causes the R8 compiler to crash during compilation [51]. In this test, *Interface Implementation* and *Reference* structures are nested in class `C0`: the inner class `C1` implements interfaces `I1` and `I2` and calls the default method `foo()` from `I1` via `super`. The crash occurs because R8 fails to resolve `foo()` in `I1` when analyzing the call `I1.super.foo()` (line 12), treats its implementation as null, and triggers a null pointer exception. R8 developers note that this bug is a partially unresolved issue from a previous bug in R8. Specifically, for the case that interfaces and classes are not nested inside `C0`, the bug is already fixed. By generating seeds with nested class and interface structures, InterFuzz exposes this incomplete fix, and the developers comment as follows: "*a special case that has not changed for a long time and has not been reported by users.*" This case shows that InterFuzz can construct complex inter-class structures and reveal subtle bugs in the compiler's analysis and optimization.

5.3 RQ2: Comparison with Existing Baselines

In this RQ, we compare InterFuzz with four baseline tools (i.e., MopFuzzer, Artemis, Jetris, and JITFuzz) in terms of both code coverage and bug detection capabilities. To ensure a fair experiment, we use the same seed pool for all tools, generated by *JavaFuzzer*.

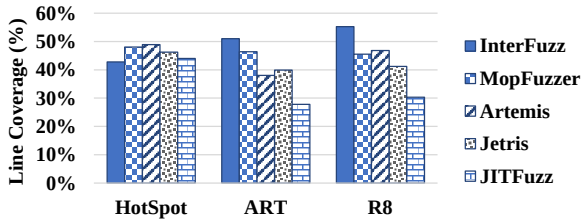


Figure 4: Comparison of line coverage

Coverage on HotSpot/ART/R8. For coverage, we run each tool for 24 hours on instrumented versions of HotSpot, ART, and R8. We average the results from three consecutive runs to ensure stability. Figure 4 shows the detailed coverage results. InterFuzz achieves 51.0% and 55.2% line coverage on ART and R8, respectively. On R8, it outperforms the best baseline, MopFuzzer, by 9.7%, and exceeds Artemis, Jetris, and JITFuzz by 8.4%, 14.0%, and 24.9%. This advantage stems from the fact that both ART and R8 place a greater emphasis on analyzing and optimizing inter-class structures. This suggests InterFuzz is better at generating complex inter-class structures, covering more inter-class-related code. Since InterFuzz is specifically designed to generate complex inter-class patterns, it effectively targets the relevant code within these compilers. In contrast, on HotSpot, InterFuzz achieves 43.2% coverage, which is lower than some baseline tools. This result is expected, as HotSpot focuses on optimizing "hot" code within methods rather than the inter-class structures that InterFuzz targets. In summary, InterFuzz generates test code that effectively targets inter-class structure handling, leading to higher coverage than the other tools.

Table 4: Comparison of bug detection ability on HotSpot, ART, and R8. The numbers in parentheses indicate the count of detected inter-class related bugs.

Compiler	InterFuzz	MopFuzzer	Artemis	Jetris	JITFuzz
HotSpot	2 (2)	3 (0)	3 (0)	2 (0)	1 (0)
ART	2 (2)	1 (0)	0 (0)	1 (0)	0 (0)
R8	3 (3)	0 (0)	1 (0)	0 (0)	0 (0)
Total	7 (7)	4 (0)	4 (0)	3 (0)	1 (0)

Bug detection ability for HotSpot/ART/R8. We run each tool for 24 hours on the debug builds of HotSpot, ART, and R8 to evaluate the ability of bug detection. Table 4 shows the number of bugs detected. Specifically, InterFuzz detects a total of 7 bugs, which involve complex inter-class structures that other tools cannot detect. On compilers focusing more on inter-class structures (i.e., ART and R8), InterFuzz outperforms others in terms of the number of detected bugs. MopFuzzer, Artemis, Jetris, and JITFuzz find 4, 4, 3, and 1 bugs, respectively. However, none of these bugs are caused by inter-class structures. Overall, InterFuzz excels compared to other tools in identifying bugs related to inter-class structures.

5.4 RQ3: Component Contribution of InterFuzz

In this RQ, we evaluate the contribution of InterFuzz's key components. We compare InterFuzz with a variant, InterFuzz$_r$, which randomly selects inter-class mutators instead of being guided by graph complexity. The comparison is based on the generated graph complexity and bug detection ability.

Graph complexity. We run InterFuzz and InterFuzz$_r$ for 24 hours and measure the graph complexity of their final seeds (i.e., the 200th generation of mutations). Figure 5 shows the comparison results. The test cases generated by InterFuzz achieve an average graph complexity of 250.8, significantly higher than InterFuzz$_r$'s average of 110.3, as confirmed by Mann-Whitney U test ($p_U < 0.05$) [30]. This result demonstrates that our complexity-guided approach is highly effective, successfully steering the mutation process toward generating programs with more intricate inter-class structures than the random strategy.

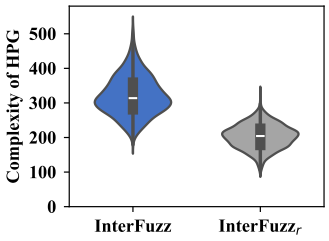


Figure 5: Graph complexity of InterFuzz and InterFuzz$_r$

Table 5: Bugs detected by InterFuzz and InterFuzz$_r$

Compiler	InterF.	InterF.$_r$
HotSpot	2 (2)	1 (0)
ART	2 (2)	0 (0)
R8	3 (3)	2 (2)
Total	7 (7)	3 (2)

Bug detection ability. We further compare InterFuzz and InterFuzz$_r$ in terms of bug detection ability. Following the same experimental setup as RQ2, we run InterFuzz and InterFuzz$_r$ on HotSpot, ART, and R8 for 24 hours each. As shown in Table 5, the results show that InterFuzz detects 7 bugs, while InterFuzz$_r$ finds only 2 bugs. This suggests that using graph complexity as guidance in the mutation process enables InterFuzz to generate test cases effectively and efficiently with complex inter-class structures.

6 Discussion

6.1 Limitation of InterFuzz

InterFuzz abstracts program into HPG and manipulates HPG by inter-class mutators to generate test cases with complex inter-class structures. Such a design offers a unified representation and eases the process of mutation. However, the variety of the generated test cases is also limited by the definition of HPG. A specific example is that InterFuzz currently only handles explicitly defined classes in the HPG and does not yet support Java's anonymous classes, resulting in a lack of ability to generate test cases containing anonymous classes. However, since we are the first to propose to validate Java compilers in terms of complex inter-class structures, our approach is novel and points out a promising direction for future research. More importantly, our evaluation demonstrates that the current design of HPG can already achieve promising performance. In the future, we plan to enrich the definition of HPG and further explore the design of more complex mutators to enhance InterFuzz's performance.

6.2 Generalizability of InterFuzz

The core design of InterFuzz is founded on two key components: the *Heterogeneous Program Graph* (HPG), which models inter-class structures, and Inter-class Mutators, which manipulate these structures. This architecture is fundamental to the framework's extensibility. Specifically, InterFuzz can be easily extended to cover a more comprehensive set of features within Java by extending the definition of HPG and mutators. For instance, to support structures not yet covered, like Java's *Annotations*, we can simply add new node or edge types to the HPG to represent them. On the other hand, InterFuzz can be efficiently adapted to other object-oriented languages like C++, Python, or Kotlin and used to test their compilers (e.g., GCC and LLVM). Because the HPG serves as a generic abstraction of program structures, higher-level components such as complexity metrics and test generation strategies remain largely language-agnostic. Porting InterFuzz to a new language mainly involves two steps: (1) mapping the new language's object-oriented constructs to the HPG and (2) implementing inter-class mutators that follow the language's syntax and typing rules. For example, in C++, we can extend the definition of HPG to support template classes, virtual functions, and other C++-specific features.

7 Related Work

We discuss related work on compiler testing.

Testing Java compilers. Fuzz testing for Java compilers is primarily divided into two categories based on the origin of test cases: generation-based and mutation-based. Generation-based methods [3, 18] create test cases from scratch using predefined models, which are often confined to fixed class structures, making it difficult to cover complex inter-class interaction scenarios and their associated bugs. Mutation-based methods [5, 6, 14, 19, 23, 54, 56, 58, 59, 61, 62] are the current mainstream, exploring new program states by modifying existing "seed" programs. In addition to works mentioned as baselines in evaluation, LeJit [58] automatically extracts templates from existing Java code and executes them to generate concrete programs for testing Java JIT compilers. JOpFuzzer [19] targets JIT compiler behavior by combining different compiler options with source code mutations. Despite their varied approaches, these techniques mainly focus on intra-procedural structures. They generally lack the ability to effectively generate and manipulate complex inter-class structures. These efforts are orthogonal to our design, offering potential for further integration to complement capabilities.

Testing other OOP languages' compilers. Compiler testing has also made significant progress for other object-oriented programming languages with complex inter-class structures, like C++ [27, 28], Python [24], and Kotlin [15, 42]. For C++ compilers, YARPGen [27] employs generation policies to guide a random program generator. This process creates test cases specifically designed to trigger scalar optimizations in C and C++ compilers. To test the Python runtime system, PYRTFuzz [24] combines generation-based program creation and mutation-based generation of application inputs. While these approaches are highly effective at testing specific compiler mechanisms, their ability to probe complex inter-class interactions within their respective languages remains limited.

8 Conclusion

We are the first to validate Java optimizing compilers with complex inter-class structures. We present InterFuzz in this study, a new fuzzing framework, which abstracts programs into *Heterogeneous Program Graph* (HPG) and guides the fuzzing by the complexity of graph. InterFuzz has identified 20 new bugs in major compilers such as HotSpot, ART, and R8. Crucially, most of these bugs are known blind spots for existing fuzzers. We believe this work can contribute to the development of more reliable Java optimizing compilers.

Acknowledgments

We sincerely thank all anonymous reviewers for their valuable feedback and guidance in improving this paper. This work was sponsored by National Natural Science Foundation of China (No.U2436207 and No.62372193).

References

[1] Android. Android open source project. https://source.android.com/, 2025. Accessed: 2025-11.

[2] Geoffrey Canright. Complex networks and graph theory. In Robert A. Meyers, editor, *Encyclopedia of Complexity and Systems Science*, pages 1244–1253. Springer, 2009.

[3] Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. Finding typing compiler bugs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2022, San Diego, CA, USA, June 13-17, 2022*, pages 183–198. ACM, 2022.

[4] Craig Chambers and David M. Ungar. Customization: Optimizing compiler technology for self, A dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation, PLDI 1989, Portland, Oregon, USA, June 21-23, 1989*, pages 146–160. ACM, 1989.

[5] Yuting Chen, Ting Su, and Zhendong Su. Deep differential testing of JVM implementations. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 1257–1268. IEEE / ACM, 2019.

[6] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun 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*, pages 85–99. ACM, 2016.

[7] Oracle Corporation and its affiliates. Openjdk. https://openjdk.org/, 2025. Accessed: 2025-11.

[8] CSmith-Project. C-Reduce, a C and C++ program reducer. https://github.com/csmith-project/creduce, 2025. Accessed: 2025-11.

[9] DemandSage. Android usage statistics 2025. https://www.demandsage.com/android-statistics/, 2025. Accessed: 2025-11.

[10] GNU Online Docs. gcov: a test coverage program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, 2025. Accessed: 2025-11.

[11] Android Source Documentation. Android runtime and dalvik. https://source.android.com/docs/core/runtime, 2025. Accessed: 2025-11.

[12] EclEmma. Jacoco java code coverage library. https://www.eclemma.org/jacoco/, 2025. Accessed: 2025-11.

[13] Google for Developers. Enable app optimization. https://developer.android.com/topic/performance/app-optimization/enable-app-optimization, 2025. Accessed: 2025-11.

[14] Tianchang Gao, Junjie Chen, Yingquan Zhao, Yuqun Zhang, and Lingming Zhang. Vectorizing program ingredients for better JVM testing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 526–537. ACM, 2023.

[15] Calin Georgescu, Mitchell Olsthoorn, Pouria Derakhshanfar, Marat Akhin, and Annibale Panichella. Evolutionary generative fuzzing for differential testing of the kotlin compiler. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, pages 197–207. ACM, 2024.

[16] Google Git. D8 dexer and r8 shrinker. https://r8.googlesource.com/r8, 2025. Accessed: 2025-11.

[17] SungHyun Hong, Jin-Chul Kim, Jin Woo Shin, Soo-Mook Moon, Hyeong-Seok Oh, Jaemok Lee, and Hyung-Kyu Choi. Java client ahead-of-time compiler for embedded systems. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2007, San Diego, California, USA, June 13-15, 2007*, pages 63–72. ACM, 2007.

[18] JavaFuzzer. Java* fuzzer for jvm. https://github.com/shipilev/JavaFuzzer, 2025. Accessed: 2025-11.

[19] Haoxiang Jia, Ming Wen, Zifan Xie, Xiaochen Guo, Rongxin Wu, Maolin Sun, Kang Chen, and Hai Jin. Detecting JVM JIT compiler bugs via exploring two-dimensional input spaces. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 43–55. IEEE, 2023.

[20] JRebel. 2024 java developer productivity report. https://www.jrebel.com/resources/java-developer-productivity-report-2024, 2025. Accessed: 2025-11.

[21] Dong-Heon Jung, Soo-Mook Moon, and Sung-Hwan Bae. Evaluation of a java ahead-of-time compiler for embedded systems. *The Computer Journal*, 55(2):232–252, 2012.

[22] Florian Latifi, David Leopoldseder, Christian Wimmer, and Hanspeter Mössenböck. Compgen: generation of fast JIT compilers in a multi-language VM. In *Proceedings of the 17th ACM SIGPLAN International Symposium on Dynamic Languages, DLS 2021, Chicago, IL, USA, October 19, 2021*, pages 35–47. ACM, 2021.

[23] Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. Validating JIT compilers via compilation space exploration. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 66–79. ACM, 2023.

[24] Wen Li, Haoran Yang, Xiapu Luo, Long Cheng, and Haipeng Cai. Pyrtfuzz: Detecting bugs in python runtimes via two-level collaborative fuzzing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 1645–1659. ACM, 2023.

[25] HeuiChan Lim and Stephen G. Kobourov. Visualizing JIT compiler graphs. In *Proceedings of the 29th International Symposium on Graph Drawing and Network Visualization, GD 2021, Tübingen, Germany, September 14-17, 2021*, pages 138–146. Springer, 2021.

[26] Linux-Test-Project. Lcov. https://github.com/linux-test-project/lcov, 2025. Accessed: 2025-11.

[27] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for C and C++ compilers with yarpgen. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–25, 2020.

[28] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Fuzzing loop optimizations in compilers for C++ and data-parallel languages. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1826–1847, 2023.

[29] Sanoop Mallissery and Yu-Sung Wu. Demystify the fuzzing methods: A comprehensive survey. *ACM Computing Surveys*, 56(3):1–38, 2024.

[30] Henry B. Mann and Donald R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[31] OpenJDK. Jdk main-line development. https://github.com/openjdk/jdk, 2025. Accessed: 2025-11.

[32] ORACLE. The java language specification - java se 17 edition. https://docs.oracle.com/javase/specs/jls/se17/html/index.html, 2025. Accessed: 2025-11.

[33] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. SPOON: A library for implementing analyses and transformations of java source code. *Software—Practice & Experience*, 46(9):1155–1179, 2016.

[34] Guillermo Polito, Stéphane Ducasse, and Pablo Tesone. Interpreter-guided differential JIT compiler unit testing. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2022, San Diego, CA, USA, June 13 - 17, 2022*, pages 981–992. ACM, 2022.

[35] Proguard. Proguard: The original java optimizer for android apps. https://www.guardsquare.com/proguard, 2025. Accessed: 2025-11.

[36] Google Git R8. Preserveinterfacemethoddispatch.java. https://r8.googlesource.com/r8/+/refs/heads/main/src/main/java/com/android/tools/r8/horizontalclassmerging/policies/PreserveInterfaceMethodDispatch.java, 2025. Accessed: 2025-11.

[37] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, Beijing, China, June 11-16, 2012*, pages 335–346. ACM, 2012.

[38] New Relic. 2024 state of the java ecosystem. https://newrelic.com/resources/report/2024-state-of-the-java-ecosystem, 2025. Accessed: 2025-11.

[39] Tim Rentsch. Object oriented programming. *ACM SIGPLAN Notices*, 17(9):51–57, 1982.

[40] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[41] Kumar Shiv, IRavishankar Iyer, Chris J. Newburn, Joakim Dahlstedt, Marcus Lagergren, and Olof Lindholm. Impact of JIT/JVM optimizations on java application performance. In *Proceedings of the 7th Annual Workshop on Interaction between Compilers and Computer Architecture, INTERACT 2003, Anaheim, CA, USA, February 8, 2003*, pages 5–13. IEEE Computer Society, 2003.

[42] Daniil Stepanov, Marat Akhin, and Mikhail A. Belyaev. Type-centric kotlin compiler fuzzing: Preserving test program correctness by preserving types. In *Proceedings of the 14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021*, pages 318–328. IEEE, 2021.

[43] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 361–371. ACM, 2018.

[44] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 361–371. ACM, 2018.

[45] JDK Bug System. Jdk7 build 147 crashed after testing my java 6-compiled web app. https://bugs.openjdk.org/browse/JDK-7064302, 2012. Accessed: 2025-11.

[46] JDK Bug System. Wrong memory state is picked in SuperWord::co_locate_pack(). https://bugs.openjdk.org/browse/JDK-8290910, 2022. Accessed: 2025-11.

[47] JDK Bug System. C2: Looplimit node is not eliminated. https://bugs.openjdk.org/browse/JDK-8260370, 2024. Accessed: 2025-11.

[48] JDK Bug System. C2: assert(can_reduce_phi(n->as_Phi())) failed: Sanity: previous reducible Phi is no longer reducible before SUT. https://bugs.openjdk.org/browse/JDK-8361699, 2025. Accessed: 2025-11.

[49] JDK Bug System. Jvm jit causes static initialization order issue. https://bugs.openjdk.org/browse/JDK-8357782, 2025. Accessed: 2025-11.

[50] Google Issue Tracker. Inconsistent execution behavior of code processed by r8 compared to d8 on hostart. https://issuetracker.google.com/issues/420228751,

2025. Accessed: 2025-11.

[51] Google Issue Tracker. R8 crash with java.lang.nullpointerexception in r8.graph.dexclassandmethod. https://issuetracker.google.com/issues/419464490, 2025. Accessed: 2025-11.

[52] Google Issue Tracker. R8 crashes when compiling the program. https://issuetracker.google.com/issues/412524379, 2025. Accessed: 2025-11.

[53] Reinhard Wilhelm, Helmut Seidl, and Sebastian Hack. *Compiler Design - Syntactic and Semantic Analysis*. Springer, 2013.

[54] Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. Jitfuzz: Coverage-guided fuzzing for JVM just-in-time compilers. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 56–68. IEEE, 2023.

[55] Yaqiang Wu, Hui Zhu, Chenyang Wang, Fujian Song, Haiping Zhu, Yan Chen, Qinghua Zheng, and Feng Tian. Programming knowledge tracing based on heterogeneous graph representation. *Knowledge-Based Systems*, 300(C):112161, 2024.

[56] Zifan Xie, Ming Wen, Shiyu Qiu, and Hai Jin. Validating JVM compilers via maximizing optimization interactions. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2024, Hilton La Jolla Torrey Pines, La Jolla, CA, USA, April 27 - May 1, 2024*, pages 345–360. ACM, 2024.

[57] Geunha You, Gyoosik Kim, Seong-je Cho, and Hyoil Han. A comparative study on optimization, obfuscation, and deobfuscation tools in android. *Journal of Internet Services and Information Security*, 11(1):2–15, 2021.

[58] Zhiqiang Zang, Fu-Yao Yu, Aditya Thimmaiah, August Shi, and Milos Gligoric. Java JIT testing with template extraction. *Proceedings of the ACM on Software Engineering*, 1(FSE):1129–1151, 2024.

[59] Zhiqiang Zang, Fu-Yao Yu, Nathan Wiatrek, Milos Gligoric, and August Shi. JATTACK: java JIT testing using template programs. In *Companion Proceedings of the 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023 Companion Proceedings, Melbourne, Australia, May 14-20, 2023*, pages 6–10. IEEE, 2023.

[60] Kechi Zhang, Wenhan Wang, Huangzhao Zhang, Ge Li, and Zhi Jin. Learning to represent programs with heterogeneous graphs. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC 2022, Virtual Event, May 16-17, 2022*, pages 378–389. ACM, 2022.

[61] Yingquan Zhao, Zan Wang, Junjie Chen, Ruifeng Fu, Yanzhou Lu, Tianchang Gao, and Haojie Ye. Program ingredients abstraction and instantiation for synthesis-based JVM testing. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, pages 3943–3957. ACM, 2024.

[62] Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. History-driven test program synthesis for JVM testing. In *Proceedings of the 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1133–1144. ACM, 2022.

[63] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Computing Surveys*, 54(11s):1–36, 2022.