# Understanding and Detecting Android R8 Compiler Bugs

ZIFAN XIE, Huazhong University of Science and Technology, China
MING WEN*, Huazhong University of Science and Technology, China
SHIYU QIU, Huazhong University of Science and Technology, China
MAOLIN SUN, Nanjing University, China
HAI JIN, Huazhong University of Science and Technology, China

Android apps currently dominate the smartphone app market, and their reliability will significantly affect user experience and even induce security risks. Since Android Gradle Plugin 3.4.0 (April 2019), the R8 compiler serves as the default infrastructure in the Android build process. It transforms JVM bytecode from the Java layer into semantic-equivalent DEX bytecode, ensuring compatibility with the Android Virtual Machine. Additionally, R8 incorporates advanced features like shrinking and obfuscation to reduce app size and enhance security. However, R8 suffers from substantial bugs, some of which can cause severe issues. This motivates us to conduct a preliminary study into R8 bugs. We collect and analyze 945 bug reports for R8. Through detailed statistical analysis, we obtain several valuable findings. For example, we identify *Optimization* as the most error-prone component. To substantiate our findings, we develop an automated testing tool named R8Scan. It utilizes a novel idea to synthesize seeds from prioritized real-world functions and construct the corresponding arguments empowered by Large Language Models (LLMs) to test R8, thus enabling the exploration of a broader range of semantics. Finally, R8Scan detects 17 R8 bugs. It also detects 10 bugs in OpenJDK and 6 bugs in Android Runtime. Among the R8 bugs, 11 are assigned priority P1, the highest developer-assigned priority level. Extensive experiments demonstrate the superiority of R8Scan over state-of-the-art JVM fuzzers. We believe this study is valuable to enhance the security of the R8 compiler.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Android R8 Compiler, Compiler Testing, Large Language Model

## 1 INTRODUCTION

Android apps currently dominate the smartphone app market, with over 2.44 million apps on the Google Play Store [49]. A key factor behind Android's widespread success is its open-source ecosystem, which promotes robust developer participation and continuous innovation. R8 compiler, the default infrastructure for the Android build process, now undertakes multiple tasks including shrinking, optimization, and obfuscation. It has been the built-in compiler since Android Gradle Plugin 3.4.0 (April 2019) [68]. A recent study [62] shows that most open-source Android apps enabled full features of R8 to build project.

*Ming Wen is the corresponding author.

Authors' addresses: Zifan Xie, Ming Wen and Shiyu Qiu are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China; emails: xzf1244@gmail.com, mwenaa@hust.edu.cn, qiusy@hust.edu.cn; Maolin Sun is with Nanjing University, China; email: merlin@smail.nju.edu.cn; Hai Jin is with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China; hjin@hust.edu.cn.


ACM 1557-7392/2026/8-ART
https://doi.org/10.1145/3837762

As the most important compiler in the Android ecosystem, R8 compiler differs from conventional compilers in several key aspects. (1) **Environment**: Unlike traditional compilers that generate executable code for specific hardware architectures, R8 focuses on transforming JVM bytecode into DEX bytecode for the Android VM. This process must account for constraints unique to the Android ecosystem, such as addressing the incompatibility between advanced Java features (e.g., 'varhandle' in Java 9) and ART. (2) **Security and Optimization**: Beyond compilation, R8 further integrates advanced shrinking, obfuscation, and other features to reduce app size and enhance app security. If these processes contain bugs, they can introduce security vulnerabilities or make apps susceptible to attacks. (3) **Compatibility Focus**: R8 is designed for Android systems of multiple versions and devices. Therefore, it employs a process called "desugaring" to transform modern JVM syntax features (e.g., *lambda expressions*) to a compatible format that can run on older Android devices. In contrast, compilers for desktop or server scenarios seldom face such extensive compatibility demands.

Due to the dominant role of R8 in the development of Android apps, R8's bugs could result in widespread unintended behaviors. In severe cases, this may even cause the app to crash at runtime [30] or induce security issues, leading to significant financial losses. To quantify the prevalence of this problem, we find that the official R8 issue tracker [55, 56] contains 2,769 reported issues as of September 2025, of which 945 have been confirmed and fixed by developers. Among them, 303 (32.1%) are assigned priority P1, the highest developer-assigned priority level in Google's issue tracker. These numbers indicate that R8 bugs are not isolated corner cases but a persistent and widespread concern. In severe cases, R8 bugs may cause apps to crash at runtime or induce security issues, leading to significant financial losses. For example, in Issue-147411673 [25], developers report that R8's incorrect optimizations make OkHttp randomly crash in an app when simultaneous requests are running. In particular, when clicking the "Make Requests" button, the app crashes after a few seconds. This issue significantly affects the stability of the app.

Despite the severity and prevalence of these bugs, no research has yet investigated the characteristics of R8 bugs. For instance, which components of R8 are more likely to contain bugs? What are the characteristics of the test cases that trigger such bugs? Answers to these questions are essential as they can guide comprehensive bug detection in R8 and provide insights for future research. Motivated by these observations, we systematically analyze the 945 confirmed bugs recorded in the official issue tracker [55, 56] following existing compiler bug studies [42, 43]. Specifically, we focus on three key perspectives:

- **Bug Priority and Symptoms.** We inspect bug priority levels and categorize bugs by their symptoms to identify common patterns. Understanding these distributions aids in designing test oracles based on symptom types.
- **Bug Distribution over Components.** By examining bug prevalence across compiler components and buggy files, we identify source files that are most prone to contain bugs. This insight helps guide directed fuzzing and resource allocation to the most problematic parts of the compiler.
- **Characteristic of Bug-Revealing Test Cases.** Users often provide artifacts to help developers reproduce the bugs. We analyze the types of artifacts that users provide. Understanding these patterns helps find bug-detection practices in real-world app development.

Through our analysis, we make several significant findings, including: (1) Among the 945 collected R8 bugs, P1-priority bugs account for 32.1% (303/945) and remain non-trivial in recent years, highlighting persistent deficiencies in R8's implementation. Notably, 360 bugs (38.1%) fall under *Miscompilation*, indicating that the DEX bytecode generated by R8 often fails to maintain semantic equivalence with the original JVM bytecode. (2) *Optimization* is the most error-prone, accounting for 39.0% bugs. Many of these bugs stem from incorrect semantic analysis or insufficient context resolution. The top three error-prone components *Optimization*, *D8*,and *Shrinking* collectively contribute to 90.8% bugs. (3) Triggering bugs requires specific code features. *Optimization* bugs depend on diverse code structures that activate its optimization passes, *D8* bugs correlate with advanced

JVM language features, and *Shrinking* bugs often stem from complex user-defined rules. This reveals the challenge of existing mutation-based fuzzing tools in effectively exposing R8 bugs.

Our empirical findings reveal unique challenges in testing R8 and directly guide the design of R8Scan. Specifically, the finding indicates that triggering R8 bugs requires highly diverse, component-specific code features (e.g., optimization-triggering structures and advanced JVM features). This presents the first challenge (C1): *how to generate test cases containing such complex structural features that existing mutation-based fuzzers struggle to produce?* Furthermore, even with the correct structural features, the bug-triggering semantics remain dormant unless specific execution paths are covered, presenting the second challenge (C2): *how to effectively instantiate arguments that explore the deep semantics of these complex structures without relying on heavy traditional symbolic execution?* To address these challenges, R8Scan incorporates two core designs. (1) Utilizing prioritized real-world code (Addressing C1). We extract *prioritized functions* from real-world open-source projects, which are then synthesized into seed programs to perform fuzzing. Unlike previous mutation-based approaches [31, 59, 63] that generate test cases based on carefully designed mutators, real-world code exhibits greater complexity and diversity in language features and control flow structures. (2) Utilizing LLMs' powerful inference capabilities to substantiate function arguments (Addressing C2). To explore the deep semantics of these *prioritized functions*, we need to substantiate several sets of arguments that can achieve high code coverage of the functions when injecting function invocations into seed files. It helps explore a wider range of semantics of the injected function. Specifically, it generates arguments using LLM-inferred argument expressions. Then, target path conditions can be satisfied when infilling caller variables that can hold any non-null value. Moreover, these expressions are computed once per function, significantly reducing the cost compared to existing LLM-based techniques that rely on repeated model interaction. Therefore, the proposed approach can enhance the diversity and effectiveness of test case generation.

We develop a differential fuzzing framework based on the above insights, R8Scan, to test R8. Specifically, we extract a large number of functions from top real-world Java projects and utilize a prioritization strategy to identify those that are more likely to trigger compiler optimizations or contain advanced JVM features. We retain such functions to activate various compilation behaviors of R8. Subsequently, we insert these prioritized functions into seeds by injecting function calls. The arguments for these functions are substantiated based on pre-generated argument expressions inferred by LLMs to synthesize effective test cases, which are then used for differential fuzzing. Ultimately, R8Scan detects 17 R8 bugs. It also detects 10 bugs in OpenJDK and 6 bugs in ART. Among the 17 R8 bugs, 11 are assigned priority P1, the highest developer-assigned priority level. Notably, all the test cases that trigger these bugs primarily originate from the diverse structural features found in real-world code, further confirming our observation. Our results demonstrate that R8Scan surpasses state-of-the-art JVM JIT fuzzers. It achieves superior code coverage and bug detection capabilities.

We summarize our key contributions below:

• **Originality:** We are the first to investigate R8 compiler bugs. We derive valuable findings from the analysis, which can aid in detecting R8 bugs and ensuring the safety of the Android ecosystem.

• **Approach:** We propose an approach to synthesizing prioritized functions from real-world projects to detect bugs. Additionally, we utilize a novel argument expression construction strategy empowered by LLMs to substantiate arguments for these functions and further devise a differential testing framework to test R8.

• **Evaluation:** We implement our approach as R8Scan. This automated testing tool detects 17 R8 bugs. It also detects 10 bugs in OpenJDK and 6 bugs in ART. Notably, 11 of the R8 bugs are assigned priority P1.

• **Open-Source:** We make R8Scan and all reported bugs publicly available. This facilitates future research. The artifacts are accessible at https://github.com/R8Scan/R8Scan.

## 2 BACKGROUND

Android apps are usually developed in JVM languages such as Java and Kotlin, but they eventually run on the Android Runtime (ART), which executes DEX bytecode rather than standard JVM bytecode. Moreover, many apps must remain compatible with older Android versions that do not directly support some modern Java language features. R8 bridges these gaps by translating JVM bytecode into DEX bytecode and rewriting feature-related bytecode patterns into forms that can execute correctly on Android devices. Beyond this translation work, R8 also performs shrinking, optimization, and obfuscation during the Android build process. Since Android Gradle Plugin 3.4.0 (April 2019) [68], R8 has served as the default compiler in the Android build pipeline.

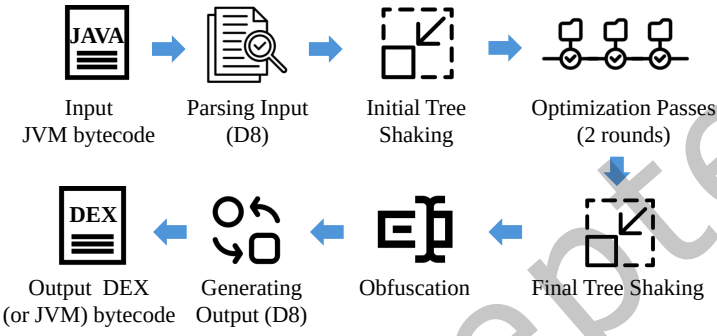



Fig. 1. Workflow of the R8 Compiler

Figure 1 illustrates the main workflow of R8 compiler, which includes the following main steps by taking JVM bytecode as input.

- **Parsing Input (*D8*)**: This process parses the input bytecode into an internal representation (IR). Because older Android versions (e.g., Android 5.0) lack native support for modern JVM language features, R8 identifies and transforms these high-level features into equivalent low-level instructions compatible with the target Android VM. This process is known as *desugaring*. For instance, `lambda` expressions are typically represented by `invokedynamic` instructions at the bytecode level. To ensure compatibility with Android 5.0, D8 rewrites these dynamic calls into static method invocations and synthetic classes, avoiding any dynamic code generation at runtime.
- **Shrinking (*Srk*)**: To reduce the application's size, R8 employs a *tree shaking* strategy. It begins by identifying all entry points of the app. From these roots, R8 analyzes the code to construct a reachability graph encompassing all resources, classes, methods, and fields that could be accessed at runtime. Any code or resources absent from this graph are deemed unreachable and are subsequently discarded to minimize the app footprint.
- **Optimization (*Opt*)**: This stage applies a wide range of advanced techniques aimed at improving runtime performance. Similar to "passes" in LLVM [21], R8 implements over 50 specific optimization passes (e.g., *Inlining* and *Loop Unrolling*) on the IR. Errors in their implementation can degrade application performance or even introduce security risks. For example, Issue-147411673 [25] was caused by an incorrect *Class Inlining* optimization, leading to a crash in the `OkHttp` network library.
- **Obfuscation (*Obf*)**: This component involves techniques such as *package flattening* and *identifier renaming*. *Package flattening* consolidates classes from multiple packages into a single one, disrupting the original code hierarchy. *Identifier renaming* translates the names of classes, methods, and variables into meaningless short strings (e.g., 'x'), significantly increasing the difficulty of reverse engineering.

- **Generating Output (*D8*)**: Finally, D8 converts the optimized and obfuscated IR into DEX bytecode for execution on ART. It can also export JVM bytecode to allow the code to run on standard JVMs. By effectively bridging JVM languages and the Android runtime environment, the generated executable can run seamlessly across diverse Android devices.

Among the four main components, *D8* provides the fundamental code transformation functionalities. To enable other components, users need to specify "`minifyEnabled true`" in the "`build.gradle`" configuration file. According to a previous study [62], 41.1% of app developers have enabled all R8 features, highlighting the importance of ensuring R8 reliability.

**R8 Bugs.** Unlike traditional compilers that mainly focus on general-purpose optimization, R8 must additionally handle Android-specific compilation requirements, such as translating JVM bytecode into DEX bytecode and rewriting modern Java/Kotlin features for compatibility with older Android versions. A concrete example helps illustrate this difference. Listing 1 shows a minimal Java program from Issue-166485528 that triggers an R8 crash. Although the source code is simple, the method reference `Target::foo` is compiled into JVM bytecode that requires special handling before it can run on older Android devices. To ensure compatibility, R8 (specifically D8) must desugar this feature into equivalent lower-level instructions supported by ART. In this issue, that compatibility-processing step caused R8 to crash during compilation. This example shows that, beyond conventional optimization, R8 must correctly process modern language features for Android compatibility, and even a small feature-related program may expose compiler bugs.

```
1  interface Target {
2    static void foo() { /* do something */ }
3  }
4  public class Test {
5    static void call(Runnable fn) {
6      fn.run();
7    }
8    public static void main(String[] args) {
9      call(Target::foo);
10   }
11 }
```

Listing 1. A small Java feature example that triggered an R8 crash (Issue-166485528).



## 3 BUG COLLECTION AND ANALYSIS

To obtain a comprehensive understanding of R8 compiler bugs, we first collect historical data from the issue trackers. Next, we aim to perform systematic investigations on the collected bugs. Following existing studies [42, 43], we focus on three key perspectives: *bug priority and symptoms*, *bug distribution over components* and *characteristic of bug-revealing test cases*.

### 3.1 Bug Collection

We collect bug reports from the two official Google issue trackers associated with the R8 project (i.e., the trackers for *R8* and *D8* [55, 56]) up to September 2025. We include both trackers because D8 is a core component in the R8 compilation pipeline, and many compiler bugs reported by developers are filed under the D8 tracker. We implement an automated crawler to retrieve all publicly accessible issue reports together with their metadata, including issue ID, title, status, priority level, report text, comments, attached reproducer, and linked fix commits

Table 1. Statistic of the Collected R8 Bugs

| Version | #Reported$_b$ | #Fixed$_b$ | #Patches | #Reproducer | Duration |
|---|---|---|---|---|---|
| 8.13.3 | 2,769 | 945 | 1,764 | 388 | July 2017-Sep 2025 |

when available. This process yields a raw dataset of 2,769 issue reports in total. We then filter the raw dataset to retain only fixed bug reports. Specifically, we retain a report if it is labeled as "Fixed" and has corresponding public fixing commits linked on the issue page. We exclude unresolved reports because they may include duplicates, intended behaviors, or feature requests. Finally, our dataset contains 945 fixed issues and 1,764 corresponding fix commits in total. We also record whether a publicly accessible reproducer (e.g., a minimal Java program or a Gradle project) is available and obtain 388 bug-revealing test cases. The remaining ones do not have a publicly accessible reproducer, mainly due to the commercial nature of Android apps. The latest R8 version is 8.13.3, and Table 1 summarizes the statistics of our collected bug reports.

**Manual Analysis and Labeling.** We follow an open-coding process [36, 47] to analyze the retained 945 fixed bugs. Two authors of this paper with experience in compiler testing first perform a pilot review on a small subset of issues (100 issues) to derive an initial coding schema for bug symptoms and associated artifacts. Based on this pilot phase, we define six symptom labels, namely *Miscompilation*, *Crash*, *Configuration Error*, *Incorrect Syntax*, *Performance Issues*, and *Error Omission*. For example, if an issue reports a "`VerifyError`" when running on Android devices, we identify it as a syntax error (classified as *Incorrect Syntax*). After the coding schema is established, two researchers independently examine each retained issue, including its report text, stack traces, comments, and linked fix commits, and assign labels accordingly. Disagreements are resolved through discussion, and a third researcher with compiler expertise joins when consensus cannot be reached directly. The inter-rater agreement measured by Cohen's Kappa is 0.85, indicating strong agreement. The final labels are determined only after full agreement is reached.

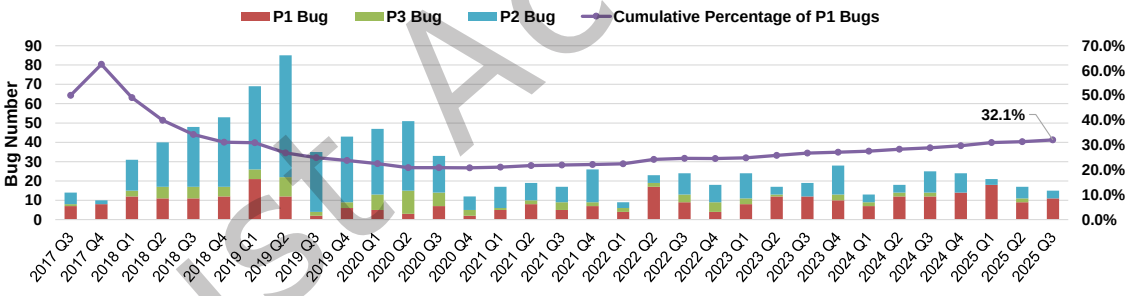



Fig. 2. Quarterly numbers of fixed R8 bugs by priority level and the cumulative ratio of P1 bugs over time. For quarter $q$, the cumulative ratio is the fraction of P1 bugs among all fixed bugs reported up to $q$.

## 3.2 Bug Priority and Symptoms

*3.2.1 Bug Priority.* In the official issue tracker, developers assign priority levels in the form of Px to indicate the urgency of a bug. Smaller numbers denote higher urgency: P1 is the highest priority, followed by P2 and P3. Figure 2 shows the quarterly number of fixed R8 bugs by priority level together with the cumulative ratio of P1 bugs over time. The initially high values (approximately 60%–80% in 2017) are expected: during R8's early development, the total number of reported bugs was small and the majority of them were P1 bugs, resulting in a high P1 ratio. As more bug reports accumulated, this ratio gradually became more stable and finally converged

to 32.1% (303 out of the 945) over the full dataset. Overall, although the total number of fixed bugs decreases after the early years, the proportion of P1 bugs remains non-trivial, which highlights the importance of detecting high-priority R8 compiler bugs.

*3.2.2 Bug Symptoms.* Error symptoms represent the observable effects of bugs during compilation or execution. We identify bug symptoms through the manual labeling process described above. Specifically, the two authors independently review each bug report (i.e., error messages, stack traces, failure scenarios, developer comments, and reproducer information) and the associated fixes (i.e., commit messages and relevant pull requests). We assign a symptom according to the primary externally observable failure: semantic mismatch for *Miscompilation*, unexpected compiler termination for *Crash*, incorrectly applied keep or configuration rules for *Configuration Error*, invalid generated bytecode or runtime verification failure for *Incorrect Syntax*, excessive compilation time or memory usage for *Performance Issues*, and missing expected diagnostics for *Error Omission*. When a bug appears to match multiple symptoms, we use the failure that is directly reported by users or confirmed by developers as the primary label. For uncertain cases, the annotators consult a third researcher with compiler expertise. The final label is recorded only after agreement is reached. We conduct a detailed review to identify the symptoms exhibited by each bug:

***Symptom 1: Miscompilation* (360 bugs, 38.1%).** This refers to cases where the output generated by R8 (i.e., DEX bytecode) does not maintain the expected semantic equivalence with the original JVM bytecode. *Miscompilation* bugs are the most common ones. This can lead to unexpected runtime behavior, such as producing incorrect outputs that deviate from the original intent. Such a bug is more severe for users compared to compiler crashes, as the program can be successfully compiled but instead behaves unexpectedly, making it harder to be detected and resolved. The previously mentioned Issue-147411673 falls under this category since the unsafe behavior exhibited in the release app. Therefore, we need a robust test oracle capable of verifying semantic equivalence.

***Symptom 2: Crash* (258 bugs, 27.3%)**. Crash bugs refer to cases where R8 terminates unexpectedly during compilation. Such bugs result in the abrupt cessation of the compilation, typically followed by error alerts. *Crash* bugs are common and critical, which prevent users' apps from being successfully compiled. Listing 2 presents a case where an R8 bug caused the Android build process to crash during IR conversion. This further exemplifies the instability introduced by R8 crashes, which can disrupt the entire build workflow, delaying the app's timely release.

```
1 $ ./gradlew clean build
2 > Task :app:minifyReleaseWithR8
3 android-sdk.jar:  NullPointerException during IR Conversion
```

Listing 2. NPE of R8 during App Build (Issue-150688800 [26])

***Symptom 3: Configuration Error* (164 bugs, 17.4%).** This refers to bugs where the developer-specified rules do not work as intended. As aforementioned, R8 replaces the previous obfuscator, ProGuard, and retains compatibility by reusing ProGuard rule formats. These rules allow developers to specify which code elements should be kept or renamed, providing fine-grained control over the final app. If R8 fails to parse and apply the rules, it can lead to unintended behavior. For instance, in Issue-130135768 [23], the developer specified the rule "`-keeppackagenames customview.extra*`" to preserve package names, but R8 still renamed the specified package to a different one, failing to adhere to the specified rules.

***Symptom 4: Incorrect Syntax* (61 bugs, 6.5%).** This symptom occurs when R8 successfully compiles DEX bytecode, but syntax errors lead to a `VerifyError` during execution by the ART. These errors prevent the app from running, causing runtime crashes.

***Symptom 5: Performance Issues* (57 bugs, 6.0%).** This denotes that the R8 compiler fails to produce results within the expected time or memory limits. Such bugs can significantly impact the developer's workflow, leading to extended build times.

***Symptom 6: Error Omission* (45 bugs, 4.8%).** This scenario occurs when R8 fails to issue an expected warning. It does not report or handle the necessary diagnostic message. For example, when a developer specifies invalid rules, R8 is expected to issue a warning. Failing to report such diagnostics leaves developers unaware of issues that could affect the final output.

**Finding 1**: Among the 945 collected R8 bugs, P1-priority bugs remain non-trivial in recent years. Moreover, 360 (38.1%) are classified as *Miscompilation*, while 258 (27.3%) cause *Crashes*. To effectively detect these bugs, a comprehensive test oracle that is capable of verifying output correctness through semantic equivalence and identifying runtime failures is necessary.

## 3.3 Bug Distribution over Components

To identify the parts most susceptible to bugs within the R8 compiler, we analyze how the 945 bugs are distributed across components. Following prior work [58], we use the source files modified by fixing commits as a practical approximation of buggy files. Although the files touched during bug fixing do not always exactly coincide with the unique root-cause files, we find that R8 bugs are often localized to a small number of files. Specifically, 50.4% of the bugs are fixed within a single file, and 73.1% are fixed within no more than five files. This high degree of localization suggests that the modified files provide a reasonable proxy for characterizing bug-prone locations in R8. By mapping patch locations to specific compiler components, we categorize each bug accordingly (e.g., Shrinking in "com/android/tools/r8/shaking"). Due to the collaborative interaction between components, a single bug may involve multiple components and source files. Therefore, the total number of bugs involving these components exceeds 945.

Table 2. Bug Location and Bug-Revealing Test Case Distribution over Four Main Components.

| Components | Optimization | D8 | Shrinking | Obfuscation |
|---|---|---|---|---|
| **#Bugs (Ratio)** | 369 (39.0%) | 319 (33.8%) | 245 (25.9%) | 87 (9.2%) |
| **#Reproducer** | 169 | 78 | 105 | 46 |

*3.3.1 Overall Distribution.* Table 2 presents the bug distribution across the R8 components. We find that *Optimization* contains the highest number of bugs, accounting for 39.0% (369/945). Following *Optimization*, *D8* has 319 bugs (33.8%); *Shrinking* has 245 bugs (25.9%). Lastly, *Obfuscation* has relatively fewer bugs (9.2%). We also found that the top 3 error-prone components (i.e., *Optimization*, *D8*, *Shrinking*) account for majority of the bug (858 bugs, 90.8%) . This suggests that testing techniques can prioritize these components.

Compared to other traditional compilers, such as LLVM and GCC, which focus on efficient compilation and optimization, features like ***Desugaring***, ***Shrinking***, ***D8***, and ***Obfuscation*** are specific to the R8 compiler. These features are closely tied to the Android ecosystem. For example, the inconsistency between the Java/Kotlin API and the Android API requires the R8 compiler to address compatibility in the generated code. These unique characteristics introduce new challenges in bug detection, which can hardly be effectively addressed by previous studies.

**Cross-Analysis of Symptoms and Components.** To further investigate how bugs manifest across compiler components, we analyzed the distribution of bug symptoms in each component. As shown in Figure 3, the symptom distributions differ across components. Overall, *Miscompilation* is the most prevalent symptom and

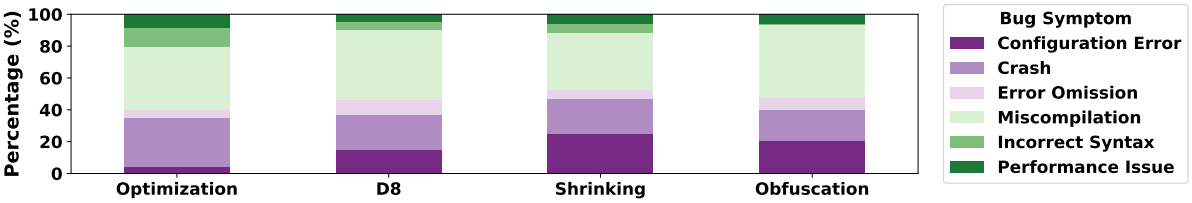



Fig. 3. Distribution of Bug Symptoms across Different Components.

Table 3. Top buggy files identified from the fixing commits of the 945 bugs

| Component | Buggy Files | #Bug | LOC | #Bug/KLOC | Description |
|---|---|---|---|---|---|
| Optimization (369 bugs) | IRConverter.java | 93 | 1,272 | 73.1 | Manages and executes IR optimization passes |
| | CodeRewriter.java | 67 | 3,972 | 16.9 | Simplifies and rewrites instructions |
| | Inliner.java | 33 | 1,392 | 23.7 | Replaces invocations with callee's body |
| | Instruction.java | 26 | 1,767 | 14.7 | Defines IR instructions for optimization |
| | MemberRebindAnalysis.java | 21 | 647 | 32.5 | Analyzes and optimizes method rebinding |
| D8 (319 bugs) | R8.java | 41 | 1361 | 30.1 | Orchestrats the overall R8 compilation |
| | DexEncodedMethod.java | 31 | 1777 | 17.4 | Represents encoded method definitions |
| | DexItemFactory.java | 23 | 3531 | 6.5 | Provides a factory to manage DEX items |
| | R8Command.java | 18 | 1660 | 10.8 | Defines and manages command of R8 |
| | DexClass.java | 16 | 1249 | 12.8 | Manages DEX class information |
| Shrinking (245 bugs) | Enqueuer.java | 103 | 5902 | 17.5 | Maintains a queue for reachability analysis |
| | ProguardConfigurationParser.java | 52 | 2435 | 21.4 | Parses Proguard configuration rules |
| | RootSetUtils.java | 38 | 2446 | 15.5 | Initials a root set for tree shaking |
| | AppInfoWithLiveness.java | 20 | 1310 | 15.3 | Encapsulates liveness info for shrinking |
| | AnnotationRemover.java | 19 | 445 | 42.7 | Removes unnecessary annotations |
| Obfuscation (87 bugs) | MethodNameMinifier.java | 18 | 430 | 41.9 | Minifies method names for obfuscation |
| | ClassNameMinifier.java | 18 | 483 | 37.3 | Minifies class names for obfuscation |
| | ProguardMapMinifier.java | 17 | 604 | 28.1 | Minifies Identifier on Proguard rules |
| | FieldNameMinifier.java | 9 | 383 | 23.5 | Minifies field names for obfuscation |
| | SourceFileRewriter.java | 8 | 112 | 71.4 | Rewrites source file attributes |



accounts for more than 36% of the bugs in every component. In particular, *D8* has the highest proportion of *Miscompilation* (44.2%), while *Optimization* is also dominated by this symptom. This suggests that bugs in these components often silently change program semantics rather than causing immediate failures. In contrast, *Crash* bugs are more prominent in *Optimization*, *Shrinking*, and *Obfuscation*.

**Finding 2**: *Optimization* is the most error-prone component (39.0% of the bugs). Moreover, the top 3 error-prone components (i.e., *Optimization*, *D8*, and *Shrinking*) are responsible for 90.8% of bugs, which suggests that testing efforts can prioritize these components.

*3.3.2 Distribution over Buggy Files.* To provide a detailed breakdown of the bugs, Table 3 further lists the top 5 buggy files in each component. For each buggy file, we present its name, the number of bugs (**#Bug**), the lines of code (**LOC**), together with a brief description. Additionally, we include the bug density metric, which is the number of bugs per thousand lines of code (**#Bug/KLOC**) following an existing study [42], which reflects the bug density among these buggy files.

**(1) *Optimization* Buggy Files.** *Optimization* analyzes and rewrites code to improve runtime performance. In each optimization pass, R8 analyzes the program to extract the context (e.g., *call chains* and *control flow*), independently checks whether the code meets the required conditions for triggering potential optimizations, and

further applies them. However, incorrect semantic analysis or insufficient context resolution can lead to bugs or security issues.

We find that *IRConverter* (93 bugs) contains the most of bugs in *Optimization*. Note that *IRConverter* is responsible for coordinating the sequence of optimization passes (e.g., Inlining) for converting IR. However, this process involves complex code transformations and the resolution of dependencies between code elements, which contributes to the high number of bugs. In *Optimization*, other files prone to error are *CodeRewriter* (67 bugs), *Inliner* (33 bugs), *Instruction* (26 bugs), and *MemberRebindAnalysis* (21 bugs). These bugs are induced by code rewriting, function inlining, IR instruction, and analyzing method rebinding. This distribution is reasonable since optimization tasks often involve complex code transformations that require accurate context and semantic analysis.

***A Case Study for Optimization.*** As shown in Listing 3, the 'IRConverter' model fails to perform correct semantic analysis and context resolution, causing a crash. The 'main' method calls 'noNormalExits' (Line 5), which throws an exception and returns abnormally. When the 'IRConverter' attempts to summarize optimization information during the pass converter process, it fails to recognize that 'noNormalExits' would return abnormally. However, the optimizer assumes that the method works normally and eventually generates incorrect optimized code. During execution, the exception thrown by 'noNormalExits' is not handled properly, thus causing the crash.

```
1  class Main{
2   public static void main(String[] args){
3    try {
4     // R8 fails to analyze callee's semantic
5     KeptClass.noNormalExits();
6     System.out.println("world!");
7     } catch (Exception e) {
8     System.out.println("Exception");
9  }}}
10 static class KeptClass {
11  public static void noNormalExits(){
12   throw new RuntimeException();
13 }}
```

Listing 3. A Test Case that Triggers a Crash During the Optimization Phase (Issue-131619590)

```
1  @interface CuAn {
2  enum MyEnum{
3   T1("Foo"), T2("Bar");
4   private final String str;
5   // R8 fails to encode param annotation
6   MyEnum(@CuAn String str){this.str = str;}
7   String getStr() {return T1 + str;}
8  }
9  public class Main {
10  public static void main(String[] args){
11   MyEnum val = System.nanoTime() > 0
12    ? MyEnum.T1: MyEnum.T2;
13   System.out.println(val);
14 }}
```

Listing 4. A Test Case that Triggers a Crash of D8 During Generating Output (Issue-235733922 [28])

**(2) *D8* Buggy Files.** D8 is a key component in the R8 compiler, where the IR is converted into DEX bytecode, aiming to ensure that the generated DEX files execute correctly on the Android VM. Typically, bugs in *D8* include non-compliant DEX formats and flaws in conversion, all of which can lead to DEX file format issues.

In *D8*, *DexEncodedMethod* (48 bugs) is the most error-prone file. In particular, it manages encoded method definitions for method-level transformations. The complexity of handling method signatures, parameter processing, and bytecode instruction conversion can lead to erroneous bytecode modifications, causing abnormal method behaviors or compilation failures. These bugs are typically caused by misunderstandings of method semantics or inadequate support for the bytecode instruction set. Additionally, other components such as *DexItemFactory* (36 bugs), *DexClass* (25 bugs), *ApplicationWriter* (15 bugs), and *DexProgramClass* (14 bugs) also contain many bugs. These bugs are primarily caused by improper DEX code management and errors in class information management.

***A Case Study for D8.*** As shown in Listing 4, R8 encounters an issue when handling the replacement of method parameter annotations, thus leading to a NullPointerException (NPE). Specifically, when encoding the constructor of the *MyEnum* enum, R8 fails to handle annotated parameters in Line 6, causing the annotation information to

be improperly cleared. This incomplete annotation handling interferes with the correct passing of parameters, disrupts the enum class's initialization process, and ultimately results in an NPE.

**(3) *Shrinking* Buggy Files.** *Shrinking* reduces the app size by removing unused code. R8 determines the entry points of an app based on user-configured rules and analyzes the code to construct a reachability graph, removing irrelevant parts. However, inaccurate code reachability resolution or improper handling of configuration rules can lead to potential bugs.

The *Enqueuer* (103 bugs) contains the most bugs in *Shrinking*, which is responsible for maintaining a queue for reachability analysis, determining which classes, methods, and fields may be accessed. This process involves complex code dependency resolution and graph construction. Any inaccuracies in analysis or mismanagement of dependencies can result in incorrect code removal, leading to semantic inconsistencies or functionality loss. Additionally, *ProguardConfigurationParser* (52 bugs), *RootSetUtils* (38 bugs), *AppInfoWithLiveness* (20 bugs), and *AnnotationRemover* (19 bugs) also have a significant number of bugs. These bugs are primarily caused by errors in improper handling of configuration rules, leading to incorrect code removal.

```
1  interface Uitf { void foo(); }
2  class UserOfUitf {
3   public static void bar(Uitf i) {/*..*/}
4  }
5  class Main {
6   public static void main(String[] arg){
7   if (System.nanoTime() < 0) {
8    // R8 fails to mark the expr as live
9    UserOfUitf.bar( ()-> System.out.println("unused") )
   ;
10   } else {
11   System.out.println("Hello");
12 }}}
```

Listing 5. The test case that triggers a crash error within Enqueuer module (Issue-232379893 [27])

```
1  abstract class Base implements Runnable {
2    abstract void foo();
3    public void run() { foo(); }}
4  //R8 fails to rename two foo as same name
5  class Sub1 extends Base {
6    void foo() { /*...*/ }}
7  class Sub2 extends Base {
8    void foo() { /*...*/ }}
9  public class Main {
10   public static void main(String[] arg) {
11   Runnable r = System.nanoTime() > 0
12     ? new Sub1() : new Sub2();
13   r.run();
14 }}
```

Listing 6. The test case that triggers a crash during obfuscation phase (Issue-133686361 [24])

***A Case Study for Shrinking.*** As shown in Listing 5, The Enqueuer module mishandled the code analysis, causing a NPE error. The lambda expression on Line 8 implements the functional interface defined on Line 1. However, R8 fails to correctly interpret the process of implementing the functional interface using the lambda expression, mistakenly identifying the contents of the lambda as unreachable and removing it. As a result, when checking the parameters in the *UserOfUitf.bar* method, it is found that the foo method of the *Uitf* interface has not been implemented, leading to a runtime error.

**(4) *Obfuscation* Buggy Files.** *Obfuscation* increases code complexity through techniques such as *identifier renaming* and *package flattening* to prevent reverse engineering. For example, *identifier renaming* changes class, method, and variable names to meaningless characters. These complex operations are prone to semantic errors and improper handling the mapping and dependencies of class/method/field before and after obfuscation.

Within *Obfuscation*, *MethodNameMinifier* (18 bugs) and *ClassNameMinifier* (18 bugs) are the most error-prone files. The former must ensure that method renaming does not break call relationships, as any mistake could cause method invocation failures. The latter must maintain inheritance and reference relationships between classes, and renaming errors can trigger class loading errors. Additionally, *ProguardMapMinifier* (17 bugs), *FieldNameMinifier* (9 bugs), and *SourceFileRewriter* (8 bugs) also involve bugs. These bugs primarily stem from misinterpretation of code structures and incorrect resolution of dependencies during the renaming process.

***A Case Study for Obfuscation:*** As illustrated in Listing 6, the test case triggers a crash during the *Obfuscation* phase. This bug primarily stems from the failure to correctly track the renaming status of certain methods during the renaming process. Specifically, the abstract method *foo* in the *Base* class is implemented in subclasses

*Sub1* and *Sub2*. A critical error occurs if the renaming process assigns divergent names to the parent method and its subclass implementations. This inconsistency breaks the method override relationship, resulting in an *AbstractMethodError*.

**Finding 3**: Analysis of the top buggy files reveals distinct bug patterns for each component. *Optimization* bugs often stem from incorrect semantic analysis during complex code transformations. *D8* bugs typically involve improper bytecode conversion and handling of method definitions. *Shrinking* bugs are commonly caused by inaccurate reachability analysis, and *Obfuscation* bugs relate to flawed dependency resolution during renaming.

### 3.4 Characteristic of Bug-Revealing Test Cases.

In this section, we analyze the publicly accessible bug reproducers. Although many bugs do not provide publicly available bug-revealing test cases, this subset can still cover all major components and is sufficient for revealing meaningful code patterns that trigger them. Based on prior researches [31, 43], test case characteristics can be analyzed from multiple perspectives. For instance, we could examine code features like the number of `for loops`. However, a more effective approach is to analyze the key factors that trigger bugs based on the unique functionality of the R8 component. In this study, we focus our analysis on the three error-prone components (i.e., *Optimization*, *D8*, *Shrinking*) according to Finding 2. These three components jointly account for 90.8% of the reported bugs. Focusing on these components allows us to cover the vast majority of bug-prone areas while keeping the analysis useful. The remaining component, Obfuscation, accounts for only 9.2% of bugs and is thus excluded from this detailed analysis.

Specifically, we observe three key factors that are likely to trigger R8 compiler bugs. **(1) Factor#1: code structures to trigger optimization passes**. *Optimization* employs 50 different passes. This involves handling complex, multi-stage analyses and transformations. Notably, these passes are only triggered when the test case includes specific code structures. **(2) Factor#2: advanced JVM features**. *D8* manages the generation of IR to ART/JVM bytecode and the parsing of input JVM bytecode. This process involves supporting many advanced JVM features. Essentially, parsing and desugaring these features requires processing their corresponding advanced instructions. Therefore, for *D8*, inadequate support for high-level JVM features can lead to format errors. A representative example is Issue-235733922 [28], in which R8 crashes because it mishandles the instructions for method parameter annotations, leading to a *NullPointerException*. **(3) Factor#3: user-defined keep rules guiding the shrinking process.** Keep rules are user-defined configuration directives (written in ProGuard rule format) that instruct R8 which program elements (e.g., classes, methods, and fields) should be preserved during shrinking and obfuscation. These rules allow developers to prevent accidental removal of critical code elements. When test cases involve complex rules, R8 may misinterpret them, leading to unexpected behavior or even crashes. A representative example is Issue-236691999 [29], where intricate *keep rules* led to the stripping of enum member, causing a *NotPresentException*.

To evaluate how such factors contribute to bug exposure, we perform a quantitative analysis of the bug-revealing test cases. As a comparison, we apply JavaFuzzer [2] to randomly generate 500 test cases and analyze their characteristics as well. We choose JavaFuzzer since it has been widely used by existing JVM testing approaches [34, 59, 63]. Then, we evaluate from the perspective of the three dimensions:

For **Factor#1**, we assess whether the 169 *Optimization*-related test cases (marked as $\text{Repro}_{Opt}$) include code structures that trigger optimizations. Specifically, R8 provides an option, "`extensiveLogg ingFilter=<func1;...>`" to print the IR changes for specified functions. After a pass, if the IR changes, the updated IR is printed; otherwise, "`Unchanged`" is printed. We measure the total number of distinct passes triggered when compiling these test cases using R8. Figure 4 shows the results. We find that each test case in $\text{Repro}_{Opt}$

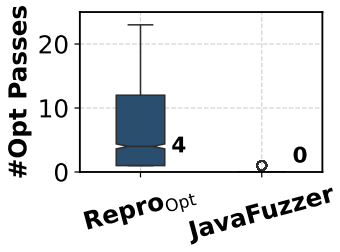



Fig. 4. Triggered Optimization Passes in Repro$_{opt}$

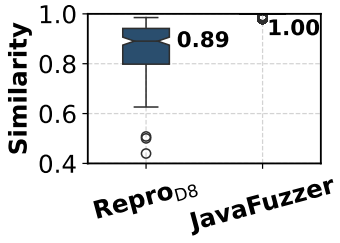



Fig. 5. The Bytecode Similarity in Repro$_{D8}$

triggers at least one optimization pass. Moreover, the full set of Repro$_{Opt}$ cases covers all 50 optimization passes in R8. In contrast, tests generated by JavaFuzzer struggle to trigger passes in R8. This suggests that detecting bugs in *Optimization* necessitates test cases with certain structures that activate optimization passes.

For **Factor #2**, we evaluate the extent to which the 78 *D8*-related test cases (marked as Repro$_{D8}$) involve advanced JVM bytecode features by analyzing their desugaring transformations. Specifically, we apply only D8 to process the JVM bytecode into two types of ART bytecode: desugared bytecode ($b_{dsg}$) and non-desugared bytecode ($b_{nodsg}$). $b_{nodsg}$ is produced via a R8 option "*–no-desugaring*". We then compute the similarity between $b_{dsg}$ and $b_{nodsg}$ using the Levenshtein Distance metric (i.e., $1 - \frac{distance(b_{dsg}, b_{nodsg})}{max(len(b_{dsg}), len(b_{nodsg}))}$), following prior work [3, 32]. Specifically, a lower similarity indicates more extensive transformations, suggesting the test case involves a richer set of advanced JVM features. Figure 5 illustrates the similarity distribution. Many test cases cluster at 0.89 or below. In contrast, test cases generated by JavaFuzzer show minimal change, as they lack advanced bytecode features. These findings suggest that incorporating complex JVM features plays a key role in triggering D8's desugaring mechanism, making such test cases valuable for testing.

For **Factor #3**, we investigate whether each bug requires specific *keep rules* to be triggered. Since R8 requires at least one *keep rule* to identify the program's entry point (i.e., the `main` method), the presence of a basic rule alone is not informative. Instead, we examine whether triggering a bug also requires additional rules that preserve other program elements. If such additional rules are present, it suggests the bug is dependent on keeping those elements. Finally, we find that among the 105 *Shrinking*-related test cases, 63 (60.0%) involved additional rules. This indicates that many bugs in *Shrinking* are sensitive to the rule set, underscoring the importance of incorporating diverse rules when generating test cases.

**Finding 4**: Triggering R8 compiler bugs requires specific code features. Bugs in *Optimization* are often triggered by diverse code structures that activate its various optimization passes. Bugs in *D8* are closely related to advanced JVM features. Meanwhile, bugs in *Shrinking* often depend on complex user-defined *keep rules*.

## 3.5 Insights and Approach Design

Our findings directly motivate the design of R8Scan. Finding 2 shows that *Optimization*, *D8*, and *Shrinking* dominate the historical bug distribution, while Finding 4 shows that bugs in these components are triggered by different kinds of code features rather than generic random mutations. These observations reveal two main challenges. First, generated tests should preserve the component-specific structures required to exercise the dominant buggy components (**C1**). Second, even when such structures are present, the bug-triggering semantics

may remain dormant unless the injected calls receive suitable arguments (**C2**). To address these challenges, we design R8Scan as follows.

**(1) Utilizing prioritized real-world code to synthesize test cases (Addressing C1).** According to Finding 4, *Optimization* bugs are often exposed by optimization-triggering structures, D8 bugs are closely related to advanced JVM features, and *Shrinking* bugs are sensitive to additional user-defined keep rules. Therefore, instead of randomly mutating seed programs or randomly harvesting code fragments, we extract standalone real-world functions and prioritize those that are more likely to exercise these component-specific behaviors. In particular, we retain functions that can activate optimization passes or contain feature patterns related to desugaring, and we augment synthesized tests with additional keep rules to better exercise *Shrinking*. We call these retained functions *Prioritized Functions*.

**(2) Utilizing LLMs to construct function arguments (Addressing C2).** When injecting functions into seed files, we aim to construct several argument sets that improve the coverage of the injected functions and better explore their semantics. Finding 1 shows that miscompilation bugs are common in R8 compilers. Such bugs do not necessarily cause crashes; instead, they surface only when the relevant execution paths are actually exercised. Traditionally, covering more branches requires symbolic execution to solve path constraints and generate argument values. However, the real-world functions collected in our study often use advanced Java syntax that existing symbolic execution tools struggle to support. In contrast, LLMs have recently shown promising capability in code inference tasks [8, 33]. Therefore, rather than using LLMs as end-to-end generators, we use them to produce argument expressions that help activate target path conditions and facilitate broader path exploration.

```
/* Argument Expression Set 1:
l=int_var1; text=String_var1.substring(0,Math.
 min(String_var.length(),9))
   Argument Expression Set 2:
l=Integer.MAX_VALUE; text=String_var1 + "
 abcdefghijk".substring(0,10) */
static String func(int l, String text) {
 StringBuilder sb = new StringBuilder();
 if (text.length() < 10) {
  try { sb.append(text, 0, l); }
  catch (Exception e) {
   System.out.println(e);
 }}
 return sb.toString();
}
```

(a) A Real-World Function from Apache Hadoop

```
public class Test {
 public static void main(String[] args){
  String s1 = "sssssssssssssss";
  int i1 = 1510218867;
  System.out.println(s1);
}}
```

(b) The Seed Program

```
public class Test {

 <place the real-world function>

 public static void main(String[] args){
  String s1 = "sssssssssssssss";
  int i1 = 1510218867;

  // Infill typed vars on argument set 1
  int l_1 = i1;
  String text_1=s1.substring(0,Math.min(s1.length(),9));
  s1 += func(l_1, text_1); // trigger the bug

  // Infill typed vars on argument set 2
  int l_2 = Integer.MAX_VALUE;
  String text_2=s1+"abcdefghijk".substring(0, 10);
  s1 += func(l_2, text_2);

  System.out.println(s1);
  }
}
```

(c) The Synthesized Test Case

Fig. 6. Example of inserting an optimization-triggering function (a) to seed (b), and generating a test case in (c). The inserted part is highlighted in orange. This triggers an *Miscompilation* bug (Issue-380182105).

**Illustrative Example.** By integrating our insights, we propose R8Scan. Figure 6 presents a motivating example of how the above insights are translated into a concrete synthesized test case. It shows a concrete example that

R8Scan inserts a real-world function to the seed program and the generated test case triggers an *Miscompilation* bug. R8Scan first extracts a real-world function from Apache Hadoop, as shown in Figure 6(a). This function is prioritized because it contains optimization-triggering structures, including a branch, exception handling, and a library call whose behavior depends on argument ranges. To activate different paths of this function, R8Scan constructs two argument expression sets in advance. One set satisfies the constraint '`text.length()<10`' and allows the try-catch branch to be executed; the other drives the library call with a large bound. Figure 6(b) shows the original seed program, and Figure 6(c) shows the synthesized test case after infilling the expressions with live variables from the seed. In particular, the first injected call uses '`text_1 = s1.substring(0, Math.min(s1.length(), 9))`', which guarantees that the branch guarded by '`text.length() < 10`' is executed. This path is crucial for exposing the bug. Theoretically, if '`l`' exceeds the length of '`text`', an `IndexOutOfBoundsException` is expected to be thrown within the function in Figure 6(a). However, during optimization, R8 incorrectly reasons that the '`StringBuilder.append(String, int, int)`' call does not require range checking and removes necessary exception-handling logic. As a result, the R8-compiled program prints a result that is semantically inconsistent with the original JVM execution. This example illustrates why both components of R8Scan are necessary: prioritized real-world functions provide bug-relevant structures, while argument expressions activate bug-relevant semantics.

We also find that such cases are difficult for existing mutation-based compiler fuzzing tools to generate. These tools typically modify seeds using optimization-triggering mutators. For instance, *JITFuzz* [59] integrates four mutators to trigger JVM optimizations, such as function inlining and escape analysis. However, these mutators do not effectively construct many bug-revealing patterns that are important for R8 (e.g., advanced JVM features and optimization-triggering code structures). Specifically, triggering Issue-380182105 requires a unique code structure involving string concatenation and exception handling. Unfortunately, the current implementations of existing tools lack support for such features.

## 4 APPROACH

The workflow is shown in Figure 7. **(1) Initialization.** This phase extracts *prioritized functions* from real-world projects. Then, we utilize LLMs to infer argument expressions to explore more code semantics for these functions, and thus effective fuzzing seeds can be further generated. **(2) Synthesizing.** This phase is pivotal, which injects the above prioritized functions into seed programs. This allows R8Scan to generate test cases that can cover a broad range of paths, thus improving the exposure of potential bugs in R8. **(3) Bug Detection and Report.** R8Scan employs differential testing to detect bugs. In particular, it utilizes *Crash* and *Miscompilation* test oracles. More importantly, this fuzzing process involves three major compilers: *R8*, *JVM*, and *ART*. Once a bug is detected, R8Scan reduces the size of test case before reporting it.

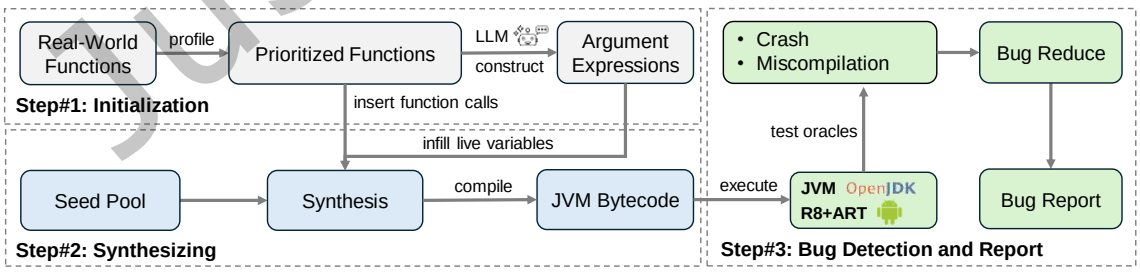



Fig. 7. Workflow of R8Scan

### 4.1 Initialization

We first extract candidate functions from real-world open-source projects and retain a subset for later test synthesis. A *prioritized function* is an extracted standalone function that is more likely to exercise bug-prone R8 behaviors identified in our empirical study. Specifically, we prioritize functions containing optimization-triggering structures, advanced JVM features related to desugaring, or patterns relevant to user-defined keep rules. We focus on this subset because not all real-world functions are equally useful for R8 testing. Our findings show that the dominant buggy components of R8 are exposed by component-specific features rather than arbitrary code fragments. Therefore, R8Scan prioritizes functions that are more likely to activate the historically error-prone components, thereby improving both testing effectiveness and generation efficiency.

**Candidate Functions Extraction.** To collect candidate functions, we select the top 500 Java projects from GitHub sorted by the number of "`Stars`". These projects are selected to ensure a diverse and high-quality set of code features to trigger various compiler behaviors. We aim to extract a comprehensive collection of real-world function database $\mathcal{D}$ from these projects. Specifically, we leverage *Spoon* [45], a well-known Java code parsing tool, to retrieve functions from source code. To facilitate the following step of seed synthesizing and fuzzing, each function in $\mathcal{D}$ should satisfy the following criteria: (a) The function must involve only *primitive types* and *built-in library types* (i.e., Java and Android library types). For example, the function can include only library function calls and should not invoke any user-defined functions. This can enable us to compile and execute functions without parsing deep classes dependencies. (b) The function must generate deterministic outputs to avoid non-deterministic operations (e.g., randomness) to facilitate differential testing. We complete the function extraction process in the following steps:

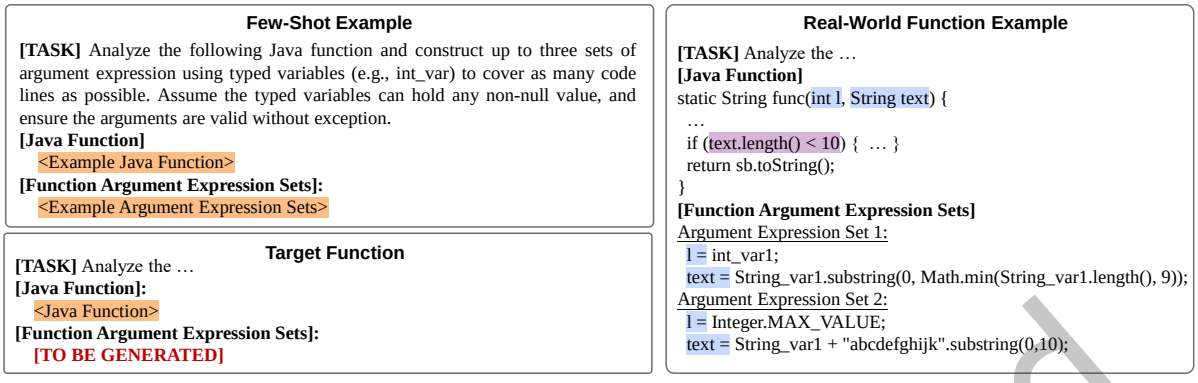
*Step 1: Remove user-defined types.* We first remove any parameter with user-defined types. We then traverse all statements following the control flows of the function, and retain a statement only if: (1) it does not reference user-defined types (e.g., user-defined functions); (2) it does not depend on a previously eliminated local variable (i.e., a parameter with a user-defined type). Additionally, if a statement references an external variable (e.g., a primitive type field), R8Scan replaces the external variable with a concrete value. Finally, if the function returns user-defined types, we replace it with `void` and rewrite all `return expr` statements as a plain `return` in the function body. The above strategy ensures that only operations involving primitive or built-in library types are retained.

*Step 2: Remove non-deterministic operations.* For each function, we randomly generate a set of arguments and execute the function ten times, logging both the return and printed values. If the results differ across executions, the function is deemed to exhibit non-deterministic behavior and is excluded from the database.

Although we adopt the above rigorous steps to filter out functions, those selected ones still retain high complexity and contain a wide range of code structures (see Section 5.1).

**Function Prioritization.** We obtain a database $\mathcal{D}$ with 82,127 functions, with the likelihood of triggering various R8 compiler behaviors varying. For instance, some contain only simple statements (e.g., *int* $x = 0$;), which would increase testing overhead without any benefit. Since we focus on *Optimization*, *D8* and *Shrinking* bugs, triggering optimization passes and containing advanced JVM features are key indicators of a function's capability.

First, we evaluate whether the function can trigger at least one pass by comparing changes in the IR before and after each pass. This aligns with the setup in Section 3.4. If a function triggers any pass, we call it an ***optimization-triggering function***. Such functions are retained in the database $\mathcal{D}_{Opt}$. Ultimately, $\mathcal{D}_{Opt}$ contains 41,230 functions. Second, we assess whether a function includes advanced language features that can activate desugaring in *D8*. If the similarity between the ART bytecodes before and after desugaring falls below 0.89, we call the function ***feature-enriched function***. We set the threshold to 0.89 since it is the median similarity of bug revealing test cases, as shown in Figure 5. Such functions are stored in database $\mathcal{D}_{D8}$, which contains 9,152 functions. All other functions that do not belong to either $\mathcal{D}_{Opt}$ or $\mathcal{D}_{D8}$ are discarded.




(a) Prompt Template for Constructing Argument Expression Sets

(b) Argument Expression Sets for Target Function

Fig. 8. Argument Expression Construction in R8Scan. (a) A prompt template, including few-show examples and function pending analysis. (b) A few examples consist of a completed prompt with the LLM-generated argument expression sets. When the expressions are infilled with variables that can take any non-null value, they are expected to satisfy the target constraints.

**Function Argument Expression Construction with LLMs.** To inject functions into seed files, we need to devise a set of arguments. In particular, we aim to construct arguments that can enhance the code coverage of the injected functions within limited calls. This helps increase the chances of detecting *Miscompilation* bugs since such bugs can only be triggered when related code is actually executed. Therefore, R8Scan leverages advanced LLMs to infer the requirements for argument expressions that trigger diverse code execution paths. This process hinges on the capability of the LLM to understand code semantics and reason about runtime behaviors. For each function, we employ few-shot in-context learning to prompt the LLM to generate multiple sets of argument expressions. Each expression consists of typed variables, such as 'int_var', combined with various operations. Figure 8(a) illustrates the few-shot prompt template used for this purpose, along with the examples provided to the LLM. The target function shares the structure of the few-shot examples but contains an empty argument field for the LLM to populate. Moreover, since a function may have many different paths, we limit the number of function calls to three at most to balance coverage exploration and execution overhead.

Take the prompt for a function in Apache Hadoop as an example. Its prompt is shown in Figure 8(b). Since the function includes the constraint '`text.length() < 10`', the LLM generates two argument expression sets. For instance, when argument '`text`' is set to '`String_var1.substring(0, Math.min(String_var1.length(), 9))`', it satisfies the above constraint. Later, during the synthesizing phase, R8Scan infills these expressions using variables with the same types from the caller (i.e., seed program). For example, when a String-type live variable (which can hold any non-null value) in the caller replaces the aforementioned `String_var1`, the constraint will be satisfied. Moreover, using the variables from caller can strengthen the contextual relationship between caller and the function. The unpredictable values from caller variables often trigger corner cases. Note that the inference of LLMs only incurs a one-time cost for each function, and the resulting argument expression sets are stored for use in later phases.

## 4.2 Seed Synthesis

In this phase, R8Scan injects prioritized functions into seed programs to construct the final test cases. The objective is to exercise as many optimization passes and advanced JVM features as possible while maximizing

code coverage within each function. Algorithm 1 outlines the general workflow of R8Scan. Specifically, the system accepts a seed corpus $\mathcal{C}$ and the enhanced function databases $\mathcal{D}_{Opt}$ and $\mathcal{D}_{D8}$ as inputs. It outputs synthesized programs for subsequent differential testing. The key steps proceed as follows:

**Algorithm 1:** R8Scan's Synthesizing Phase

```
   Input  : A seed corpus C; The enhanced function database D_Opt and D_D8
1  p ← getSeed(C) // Obtain a seed file from C
2  for 1..MAX_ITER do
3      // Randomly process a function from D_D8 and D_Opt
4      p ← Synthesize(p, choose(D_opt))
5      p ← Synthesize(p, choose(D_D8))
6  end
7  Function Synthesize(p, func):
8      pos ← p // Randomly select a line pos from p
9      {Arg_set} ← loadLLMResult(f) // Load pre-generated argument expression sets from LLM
10     foreach Arg_set ∈ {Arg_set} do
11         inp_1, inp_2, ..., inp_n ← getArg(pos, p, func, Arg_set) // Create arguments by reusing live variables or generating
               new variables to infill the expression
12         ret ← getReturn(pos, p, func) // Try to assign the return value to a live variable
13         p ← insert(inp_1, inp_2, ..., inp_n, ret, pos, p) // Insert the function call and update program p
14     end
15     return p
16 end
```

(1) Randomly select a seed program $p$ (Line 1).

(2) Randomly process a function from $\mathcal{D}_{D8}$ and $\mathcal{D}_{Opt}$ respectively, until reaching the maximum number of iterations. In R8Scan, we set `MAX_ITER` to 20 to achieve a trade-off between performance and overhead (Line 2-6).

(3) Randomly pick a random position *pos* for injection (Lines 8) and load the function argument expression sets pre-generated from LLMs (Line 8-9).

(4) For each argument expression set, we substantiate the arguments ($inp_1, inp_2, ..., inp_n$) either by reusing live variables or generating new ones to infill the expression. For example, if the expression requires a *String* type variable, we randomly select a *String*-type live variable from the caller to infill the expression. We also randomly select a live variable to receive the return value, ensuring the changes caused by the function are observable, as such changes can be reflected by printing the variable's value (Lines 11-12).

(5) Insert a function call into $p$ at *pos* using the selected inputs and return variable. It updates $p$ for the next iteration (Line 13).

We maintain syntactical and semantic correctness during function injection. For the injected functions, we import the necessary Java and Android libraries into the seed file during function injection. Regarding the generated arguments, if a function requires arguments of built-in types, the LLM generates matching expressions. When an expression needs a library object that is not present in the seed code, R8Scan creates a new one based on its constructor. For example, R8Scan might generate `new java.util.Random(100)` to create a Random object according to the constructor `Random(long seed)`. Section 3.5 has illustrated how these designs work together in a concrete example.

### 4.3 Bug Detection and Report

**Bug Inspection**. After generating test cases, we need to specify rules as the entry points of the program. Finding 4 reveals that triggering bugs in *Shrinking* often relies on complex rules. Therefore, R8Scan randomly specifies rules to keep the inserted functions beyond the `main` function. As a result, all methods that are unreachable from the entry points are removed, which helps in detecting *Shrinking* bugs. Subsequently, R8Scan identifies bugs using the following test oracles:

- *Crash*: First, R8Scan uses *Javac* to compile the test case into JVM bytecode. Then, R8Scan applies *R8* to compile the JVM bytecode into DEX bytecode. A bug is detected if *R8* crashes at runtime.
- *Miscompilation*: R8Scan uses JVM to execute the JVM bytecode and record the output $O_{JVM}$. Similarly, we specify options for JVM to enable JIT compilation, aiding the detection of JVM JIT bugs. We then execute the DEX bytecode on ART and record its output as $O_{ART}$. We compare $O_{ART}$ with $O_{JVM}$ to detect semantic inconsistencies. However, inconsistencies might originate from any of the three compilers: *R8*, *JVM*, or *ART*. To pinpoint the source of such discrepancies, we manually investigate to determine which compiler is responsible.

**Test Case Reduction**. R8Scan frequently generates complex test cases by injecting diverse real-world functions into seeds. Consequently, these test cases expand in size and invoke a wider range of compiler behaviors. However, this increased complexity hinders developers from identifying the root cause of defects. To facilitate the debugging process, we employ the state-of-the-art tool ***perses*** [51] to minimize test cases prior to reporting bugs.

## 5 EVALUATION

We evaluate R8Scan with the following research questions:

- **RQ1 (Usefulness):** Can R8Scan successfully detect new bugs in the R8 compiler?
- **RQ2 (Effectiveness):** Does R8Scan outperform state-of-the-art techniques in terms of code coverage and bug detection capability?
- **RQ3 (Components' Contribution):** How do the major design components contribute to the effectiveness of R8Scan? We also evaluate the performance variations across different backbone LLMs.

### 5.1 Experiment setup

**Target R8, JVM and ART.** For R8, we compile its latest version (*commit*=`740f37`). For JVM, we select one of the most popular JVM implementations, OpenJDK [44], as our test target. We compile the latest debug build of the mainline version (OpenJDK 25) and configure it with the `-Xcomp` flag to mandate JIT compilation. Regarding ART, we use the latest version of AOSP [19] (Android Open Source Project) as our target. We compile ART from the latest AOSP source and specify the '`-Xjitthreshold:0`' option to enforce JIT compilation.

**Environment.** We conduct our evaluation on a Linux server equipped with two Intel(R) Xeon(R) Gold 6248R CPUs and 256GB of RAM. To construct function argument expressions with LLMs, we choose DeepSeek-V3 [13] since it has been proven to exhibit superior performance in code-related tasks [20]. It is worth noting that we also tried other LLMs, but found that DeepSeek-V3 helped the tool achieve higher coverage and discover more bugs. Consequently, we chose DeepSeek-V3 (see Section 5.4.1 for details). The prompt templates can be found in our repository. For each function in $\mathcal{D}_{Opt}$ and $\mathcal{D}_{D8}$, we prompt the LLM to generate expressions with the temperature set to zero via the DeepSeek APIs. This LLM-based preprocessing stage incurs a one-time cost of $6.27 in total, corresponding to 1.2 million input tokens. The entire inference phase took approximately 4 hours of wall-clock time, and the generated expressions were saved and reused throughout the subsequent fuzzing process. For code coverage comparison, we use the well-known Java coverage analysis tool *Jacoco* [1] to measure the coverage.

**Seed Pools.** Following existing studies [31, 63], we obtain the initial seed pool using *JavaFuzzer* [2].

**Prioritized Function Pools.** We quantify the number of lines and branches for the functions residing in databases $\mathcal{D}_{Opt}$ and $\mathcal{D}_{D8}$ (see Section 4.1). Figure 9 presents the results. We observe that the majority of functions consist of fewer than 60 lines and 15 branches. On average, a function contains 15.07 lines and 2.95 branches. The number of lines and branches in a function often correlates with its error risk. Complex functions can make it more difficult for compiler to achieve correct compilation. Hence, such functions are non-trivial and useful for detecting bugs.

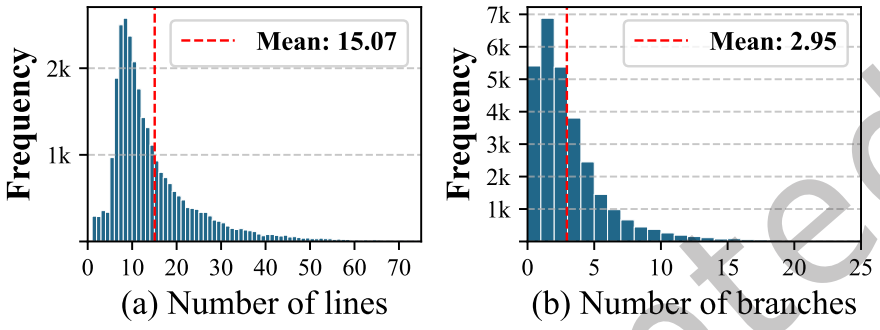



Fig. 9. Distribution of Lines and Branches in Functions

Table 4. Covered Unique Paths by Argument Expression Sets

| #Unique Paths | #Functions | |
|---|---|---|
| | size(AES)=3 | size(AES)=2 |
| 3 | 9.6K, 85.7% | - |
| 2 | 1.0K, 9.0% | 25.1K, 89.6% |
| 1 | 0.6K, 5.3% | 2.9K, 10.4% |

**The LLM-inferred Argument Expression Sets.** We further evaluate whether the expressions inferred by LLMs help explore different paths in the functions as expected. For each prioritized function, R8Scan infers at most three Argument Expression Sets (AES), so that the injected function is invoked no more than three times in a synthesized test case. We do not assume these LLM-inferred expressions are always perfect; instead, we later assess whether they effectively guide distinct path exploration. Suppose that the LLM generates three AES for a function. We use these AES to create three sets of arguments and invoke the function three times. Each typed variable in the expression is assigned a random value. Next, we count how many distinct paths are covered across these three executions. Executions that cover the same lines of code (measured by JaCoCo's coverage data) are considered the same path. In our analysis, we exclude functions whose AES size is 0 or 1, since these functions contain only a single path. There are 46,159 distinct functions in $D_{Opt}$ and $D_{D8}$. Table 4 shows the results. The statistics indicate that the expressions generated by LLMs can effectively cover multiple paths in most cases. For example, when the AES size is 3, 9.6K functions (85.7%) cover three distinct paths with three calls. These results suggest that LLMs can effectively guide broader path exploration.

**Baselines.** R8Scan is the first tool specifically designed for testing R8. We evaluate its bug detection capability by comparing it with three leading JVM JIT fuzzers: *JITFuzz* [59], *Artemis* [34], and *Jetris* [71]. Note that these tools are designed for testing JVM JIT compilers, not R8. In particular, *Artemis* mutates seeds at the source code

Table 5. Statistic of Bugs Detected by R8Scan. 'Detection' indicates if any baselines can detect the bug

| Bug ID | Compiler | Symptoms | Affected Component | Status | Priority Level | Detection |
|---|---|---|---|---|---|---|
| Issue-341618078 | R8 | Miscompilation | Optimization | Fixed | P1 | ✗ |
| Issue-342067836 | R8 | Miscompilation | Optimization | Fixed | P1 | ✗ |
| Issue-344363462 | R8 | Miscompilation | Optimization | Fixed | P1 | ✗ |
| Issue-348499741 | R8 | Miscompilation | D8 | Fixed | P1 | ✗ |
| Issue-354625681 | R8 | Miscompilation | Optimization | Fixed | P1 | ☒ |
| Issue-354625682 | R8 | Miscompilation | Optimization | Fixed | P1 | ☒ |
| Issue-358913905 | R8 | Crash | Optimization | Fixed | P1 | ✗ |
| Issue-418568424 | R8 | Crash | D8 | Fixed | P1 | ✗ |
| Issue-369739224 | R8 | Miscompilation | D8 | Fixed | P1 | ✗ |
| Issue-371247958 | R8 | Miscompilation | Optimization | Fixed | P1 | ✗ |
| Issue-384844007 | R8 | Miscompilation | Optimization | Fixed | P1 | ✗ |
| Issue-379347949 | R8 | Miscompilation | D8 | Fixed | P2 | ✗ |
| Issue-380109542 | R8 | Miscompilation | Optimization | Fixed | P2 | ✗ |
| Issue-380182105 | R8 | Miscompilation | Optimization | Fixed | P2 | ✗ |
| Issue-359102835 | R8 | Crash | Optimization | Fixed | P3 | ✗ |
| Issue-353143650 | R8 | Miscompilation | Shrinking | In Progress | P2 | ✗ |
| Issue-370217723 | R8 | Miscompilation | Optimization | In Progress | P3 | ✗ |
| Issue-350540996 | ART | Miscompilation | JIT Optimization | Fixed | P2 | ✗ |
| Issue-351868772 | ART | Crash | JIT Optimization | Fixed | P2 | ✗ |
| Issue-368984521 | ART | Miscompilation | JIT Optimization | Fixed | P2 | ✗ |
| Issue-341476044 | ART | Miscompilation | JIT Optimization | Fixed | P2 | ✗ |
| Issue-370303498 | ART | Miscompilation | LibCore | In Progress | P3 | ✗ |
| Issue-347706992 | ART | Miscompilation | JIT Optimization | Duplicate | P2 | ✗ |
| JBS-9077400 | OpenJDK | Crash | JIT C2 Compiler | Fixed | P2 | ✗ |
| JBS-9077067 | OpenJDK | Crash | Javac | Fixed | P4 | ✗ |
| JBS-9078164 | OpenJDK | Crash | JIT C2 Compiler | In Progress | P4 | ✗ |
| JBS-9077257 | OpenJDK | Miscompilation | JIT C2 Compiler | Duplicate | P2 | ✗ |
| JBS-9077247 | OpenJDK | Miscompilation | JIT C2 Compiler | Duplicate | P2 | ✗ |
| JBS-9077393 | OpenJDK | Crash | JIT C2 Compiler | Duplicate | P2 | ✗ |
| JBS-9077215 | OpenJDK | Crash | JIT C2 Compiler | Duplicate | P2 | ✗ |
| JBS-9077593 | OpenJDK | Miscompilation | JIT C2 Compiler | Duplicate | P3 | ✗ |
| JBS-9077754 | OpenJDK | Crash | JIT C2 Compiler | Duplicate | P4 | ✗ |
| JBS-9077755 | OpenJDK | Crash | Javac | Duplicate | P4 | ✗ |

level to generate mutants, while *JITFuzz* and *Jetris* perform mutation at the Java bytecode level. To include these tools for comparison, we select mutants generated by them for differential testing and reuse the three test oracles mentioned in Section 4.3. For fairness, we assign the same *JavaFuzzer*-generated seed pool to all baselines.

## 5.2 RQ1: Usefulness of R8Scan

**Detected Bugs.** We apply R8Scan to fuzz the latest version of R8 as the main target, and use OpenJDK and ART as additional generalization targets. R8Scan detects 17 R8 bugs. Table 5 presents a detailed breakdown of the bug statistics. Through manual root cause analysis, we find that *Artemis* also detects two of the R8 bugs (Issue-354625682 and Issue-354625681). However, the remaining R8 bugs remain undetected by the baselines because the key code structures that trigger them originate from real-world functions, which represent patterns that are challenging for baselines to synthesize (see Section 5.3). Among the 17 R8 bugs, 14 are classified as *Miscompilation*. Such bugs pose significant risks because they do not cause compiler crashes, which makes them difficult to detect. Regarding priority levels, 11 R8 bugs are assigned priority **P1**. Moreover, there are 4 R8 bugs with P2 and two with P3. This indicates that the R8 bugs detected by R8Scan are practically important. Since these bugs could cause crashes during the build process, they might delay the app's release if such bugs are not detected and fixed. R8Scan also detects 10 bugs in OpenJDK and 6 bugs in ART. Moreover, fixing our detected bugs is non-trivial. For example, ART developers comment in Issue-341476044, "*We are still working on this issue. The fix is harder than we originally thought.*" Considering the importance of OpenJDK and ART in the Java and

Android ecosystems, detecting their bugs is crucial for ensuring the stability, performance, and security of Java and Android software.

**Affected Components.** The 17 detected R8 bugs span multiple R8 components, as detailed in Table 5. The *Optimization* component accounts for the majority of these R8 defects (12/17). That is rational since *Optimization* is the most error-prone component according to our Finding 2. Additionally, R8Scan detects 4 bugs in *D8* and 1 bug in *Shrinking*. Regarding JVM and ART, R8Scan also identifies bugs related to *Optimization*. For example, there are 8 bugs in the JVM JIT C2 compiler, and 5 bugs affect *Optimization* in ART. This demonstrates the effectiveness of R8Scan in detecting bugs by injecting prioritized real-world code.

### 5.3 RQ2: Effectiveness of R8Scan

We evaluate R8Scan against the baselines using the same seed pool *w.r.t.* achieved *code coverage* and *bug detection capability*. Regarding code coverage, we focus on the four main components of R8. To mitigate the effects of randomness, we repeat each experiment three times and report the average values. For tools that employ iterative mutation to generate test cases, we select the final output for differential testing. For instance, we utilize the $1,000th$ test case from *JITFuzz*, as it performs 1,000 mutation iterations per seed. It is worth noting that seed processing throughput varies across tools. To ensure fairness, we standardize the fuzzing duration at 24 hours for all tools.

**Code Coverage.** Figure 10 depicts the code coverage results averaged across multiple experimental runs. The data reveals that R8Scan surpasses all baselines in terms of line coverage. Specifically, it attains 49.8% line coverage across the four main components, whereas *JITFuzz*, *Artemis*, and *Jetris* achieve 29.6%, 40.6%, and 42.5%, respectively. Notably, the four main components of R8 comprise approximately 406.5K lines of code. Consequently, even a 1% improvement in code coverage can translate to thousands of additional lines being tested. The superior code coverage achieved by R8Scan demonstrates that our focus on using real-world code as input seeds is effective for detecting deeper bugs. In *Optimization*, R8Scan significantly outperforms other tools, achieving 65.9% coverage. In contrast, *JITFuzz* covers only 32.1%, while *Artemis* and *Jetris* attain 51.9% and 53.9%, respectively. This substantial gap indicates that R8Scan triggers a broader range of code related to optimization processes, showcasing its potential to uncover more compiler optimization bugs. For other modules, such as *Shrinking* and *D8*, R8Scan maintains high coverage, achieving coverage of 45.9% and 47.4%, respectively. In conclusion, the test cases generated by R8Scan induce R8 to trigger more optimization passes, thereby achieving higher code coverage.

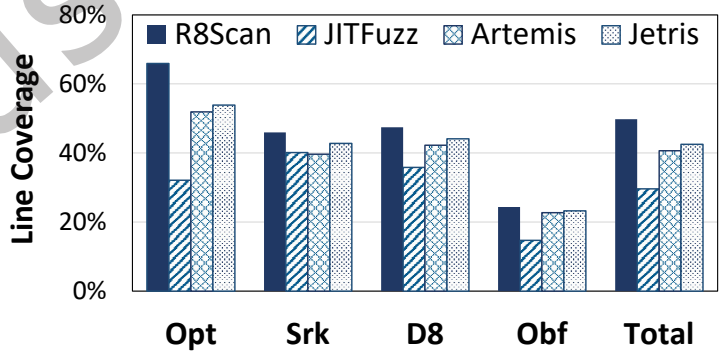



Fig. 10. Comparison of Line Coverage in R8

**Comparison of Bug Detection Capability.** We assess the bug detection efficacy of R8Scan relative to the baselines by using a shared seed pool to count the bugs found within a 24-hour duration. As shown in Table 6, R8Scan detects a total of 7 bugs. *JITFuzz*, *Artemis* and *Jetris* can detect 1 bug, 4 bugs, and 2 bugs, respectively. We find that R8Scan can detect 4 bugs in R8. In contrast, the best-performing baseline, *Artemis*, is able to detect only 1 bug in R8. We also examined whether the baselines expose bugs missed by R8Scan. According to our root-cause analysis, we did not observe any baseline-unique bugs that were completely missed by R8Scan. Therefore, R8Scan outperforms the baselines in terms of bug detection.

Table 6. Comparison of Bug Detection Capability. Numbers in parentheses indicate unique bugs detected by each tool through manually checking the root causes.

| **Compiler** | R8Scan | **JITFuzz** | **Artemis** | **Jetris** | **JavaFuzzer** |
|---|---|---|---|---|---|
| R8 bugs | 4 (3) | 0 (0) | 1 (0) | 0 (0) | 0 (0) |
| JVM bugs | 2 (2) | 1 (1) | 3 (3) | 2 (2) | 0 (0) |
| ART bugs | 1 (1) | 0 (0) | 0 (0) | 0 (0) | 0 (0) |
| **Total** | **7 (6)** | **1 (1)** | **4 (3)** | **2 (2)** | 0 (0) |

Through injection of prioritized functions via LLM-aided semantic exploration, R8Scan demonstrates high effectiveness in uncovering complex bugs. Conversely, the baselines focus on a more limited spectrum of optimization strategies. Specifically, *Artemis* concentrates on loop-related mutators to verify different JIT compilation settings. These mutators render specific code parts "hot" to dictate their selection for JIT compilation. During this process, *Artemis* also injects code snippets from JVM regression tests, but these snippets are limited in diversity compared to real-world code. Additionally, the employed rule-based injection strategy fails to explore the rich semantics of the real-world code as it can only cover limited execution paths. *Jetris* primarily learns bug-revealing patterns from historical test cases but fails to incorporate structurally richer real-world code. *JITFuzz*, on the other hand, focuses only on escape analysis, function inlining, and simplification. As a result, a broader range of optimization passes is not sufficiently analyzed, leading to suboptimal bug detection.

## 5.4 RQ3: Contribution of the Major Designs

The key designs of R8Scan are (1) the prioritization strategy to retain only the optimization-triggering and feature-enriched functions; (2) function argument expression inferred by LLM. We develop the following variants to evaluate R8Scan based on four key metrics: bug detection capability, triggered optimization passes, achieved bytecode similarity, test case code coverage. For a fair comparison, when injecting a function, we ensure that the number of injected function calls matches the size of argument expression sets of that function. Other settings are aligned with RQ2 (i.e., the same seed pool and a testing time).

- $\text{R8Scan}_o$: This variant randomly inserts 40 functions selected from all functions extracted from real-world projects, thus examining the effectiveness of the prioritization strategy. This variant helps determine whether our prioritization strategy can accelerate bug detection.
- $\text{R8Scan}_p$: This variant disables argument expressions generated from LLM. It randomly reuses live variables or creates new variables to form arguments. This variant helps determine whether the argument construction strategy empowered by LLM is effective. Specifically, we measure the coverage of the synthesized test cases to reflect to what extent the semantics of the injected real-world functions have been explored. Higher coverage indicates more thorough exploration.

**Results Analysis.** (1) For bug detection capability, the results are shown in Figure 11(a). As time progresses, R8Scan is able to uncover more bugs. In contrast, $\text{R8Scan}_o$ (4/7) and $\text{R8Scan}_p$ (5/7) detect fewer bugs. This shortcoming stems from $\text{R8Scan}_o$'s approach to generating new test cases. Its function database contains many

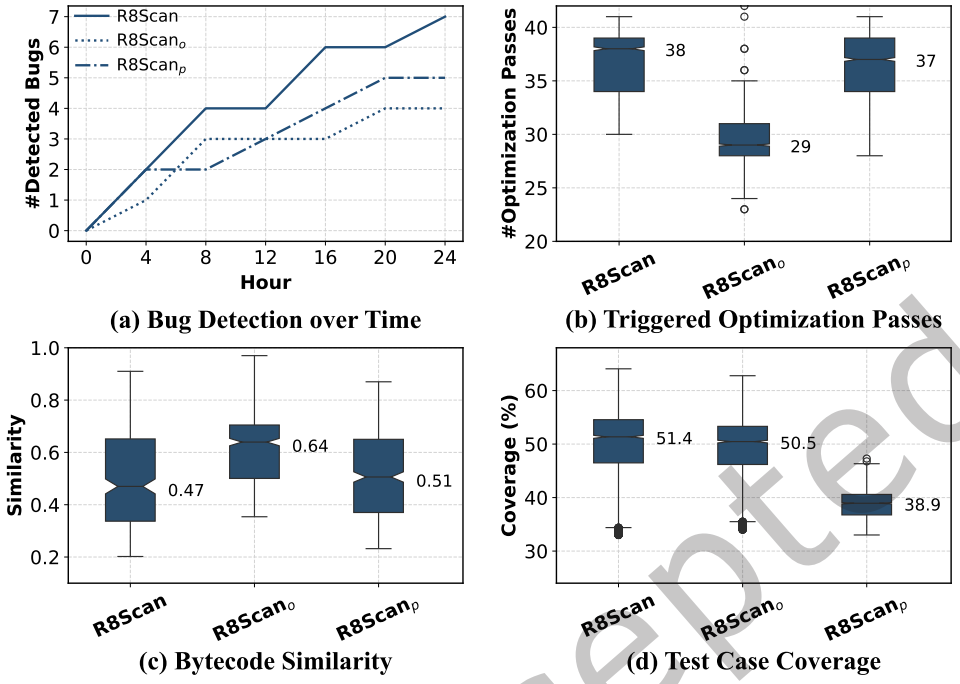



Fig. 11. Comparison of R8Scan and Its Variants.



ordinary functions that are ineffective in triggering optimizations. As a result, within the same timeframe, R8Scan$_o$ wastes resources executing these ordinary functions, increasing overhead. (2) For triggered optimization passes, the results are shown in Figure 11(b). Concretely, R8Scan outperforms its variants by triggering more passes. In contrast, R8Scan$_o$, which utilizes all functions extracted from real-world projects, shows a noticeable drop in the median (23.7% = (38-29)/38). This is because R8Scan$_o$'s function database contains many ordinary functions that are ineffective in triggering optimizations. (3) For achieved bytecode similarity, R8Scan can achieve lower bytecode similarity than R8Scan$_o$). This is because R8Scan employs feature-enhanced functions that incorporate more JVM features, thereby activating more desugaring operations in D8. (4) For test case code coverage, the results are shown in Figure 11(d). We can find that R8Scan outperforms R8Scan$_p$ by covering 12.5% (51.4%-38.9%) more code lines. This is because R8Scan uses LLMs to construct arguments for injected function calls. In contrast, R8Scan$_p$ randomly reuses variables or generates new ones to create arguments, often failing to satisfy various constraints in each function, which leads to lower code coverage. Overall, such results reflect that our ideas of utilizing prioritized real-world functions and argument expression construction strategy can drive R8Scan to detect R8 bugs effectively.

*5.4.1 Robustness across LLMs.* We evaluate the performance of R8Scan across different language models for checker synthesis. In addition to our default model, DeepSeek-V3, we test GPT-4o and Gemini-2.5-flash to generate the AES for each function. The experiment is conducted on the same function database, and we measure the impact of each LLM on both bug detection capability and average test-case code coverage over a 24-hour period.

**LLM Selection.** As detailed in Table 7, our default choice, DeepSeek-V3, yields the best results, detecting 7 bugs and achieving the highest average test-case coverage at 51.4%. GPT-4o also performs strongly, detecting 6 bugs

with 48.3% coverage, while Gemini-2.5-flash detects 5 bugs with 49.1% coverage. We attribute the advantage of DeepSeek-V3 to its stronger code reasoning capability. Our inspection of sampled outputs shows that DeepSeek-V3 yields fewer syntactically invalid/type-incompatible expressions than GPT-4o/Gemini, which improves path coverage and bug detection.

Table 7. The Impact of Different LLMs on R8Scan's Performance

| Variants | #Detected Bugs | Average Test Case Coverage |
|---|---|---|
| W/ DeepSeek-V3 (default) | 7 | 51.4% |
| W/ Gemini-2.5-flash | 5 | 49.1% |
| W/ ChatGPT-4o | 6 | 48.3% |

# 6 DISCUSSION

## 6.1 Threat to validity

The primary threat to **external validity** lies in the effectiveness of argument expressions constructed by LLMs. If these expressions fail to guide diverse path exploration, they might miss the semantics of injected functions. To minimize this threat, we validated their effectiveness, proving LLMs can support broader path exploration (see Table 4). Besides, our experiments show this strategy can enhance test case coverage and R8 bug detection (see Section 5.4). More importantly, this strategy incurs only a one-time cost per function, and the resulting expression sets can be reused, significantly cutting costs compare to inferring arguments during fuzzing.

A key threat to **internal validity** is our focus on three specific characteristics (optimization-triggering code structures, advanced JVM features, and complex *keep rules*) within bug-revealing test cases. In fact, triggering a bug also depend on other factors or contexts. For example, triggering Issue-380182105 requires not only code structure to trigger optimizations but also specific context. However, the three factors are directly tied to the core functionality of R8's most error-prone components. Our quantitative analysis further confirms a strong correlation between these factors and historical bugs. This suggests they are key factors of bugs, making our findings a valid basis for guiding future testing efforts.

## 6.2 The Generalizability of R8Scan

Many modern compilers optimize the code to boost execution efficiency. For example, there are GCC [18] and LLVM [21] for C/C++, rustc [14] for Rust, CPython [11] for Python and V8 [39] for JavaScript. The fundamental concepts of R8Scan generalize to these compilers for the purpose of bug detection. To test other compilers (e.g., rustc and V8), we can extract function-level snippets from representative projects and inject these calls into the seed inputs. This method helps cover diverse language features. However, some parts of R8Scan are still specific to R8. In particular, our prioritization strategy is based on R8 bug characteristics, and our differential testing workflow is designed for the R8-to-ART compilation pipeline. Therefore, adapting R8Scan to other compiler ecosystems still requires some ecosystem-specific changes, such as incorporating language-specific constructs and compiler-specific execution oracles into the workflow.

# 7 RELATED WORKS

**Understanding Software Bugs.** Understanding bug characteristics is fundamental to various software engineering tasks, notably automated testing [6, 15, 41, 50, 70] and debugging [7, 51, 54, 65, 69]. Empirical studies on bugs primarily aim to investigate and extract these characteristics to enhance comprehension of the studied bugs. Some research focuses on specific types of bugs. For instance, Xu *et al.* [64] investigates the silent security bugs

induced by compilers. Lu *et al.* [41] describe the characteristics of concurrency bugs. Dietz *et al.* [15] examine integer overflow bugs. Additionally, some studies analyze bugs in specific real-world software systems. Jia *et al.* [22] analyze bug patterns in the TensorFlow system. Garcia *et al.* [17] investigates bugs in autonomous driving systems. In contrast, this paper focuses on R8, a critical infrastructure component in the Android ecosystem, and presents the first large-scale empirical study of its bugs.

**Compiler Testing.** Compiler testing has been extensively studied across different execution environments and programming languages [9, 72, 73]. For managed-language runtimes, Chen *et al.* [10] propose deep differential testing of JVM implementations by systematically generating test programs and comparing execution results across multiple JVMs. JITFuzz [59] introduces coverage-guided fuzzing for JVM JIT compilers with carefully designed bytecode-level mutators to trigger optimization bugs, while SJFuzz [60] further improves JVM fuzzing efficiency through seed and mutator scheduling. For JavaScript engines, Wang *et al.* [57] employ input encapsulation templates to trigger JIT compilation and enable self-checking test cases, and Bernhard *et al.* [4] exploit the interplay between the interpreter and the JIT compiler to uncover bugs.

Compiler testing has also been widely explored for other language ecosystems such as C/C++ and Rust [5, 16, 35, 37, 38, 40, 48, 52, 53]. For example, Aamodt *et al.* [35] compare compilation outputs across multiple compilers to identify unstable code caused by undefined behaviors. Theodoridis *et al.* [52, 53] leverage dead code elimination to analyze optimization effects and uncover missed optimization opportunities. In the Rust ecosystem, Sharma *et al.* [48] develop a random program generator that respects Rust's type, borrowing, and lifetime rules for deterministic compiler testing.

Compared with these studies, R8Scan differs in three important aspects. First, it targets the Android R8 compiler, which contains unique components such as desugaring, shrinking, and obfuscation that are absent from conventional JVM and JavaScript JIT compilers. Second, rather than relying solely on mutation-based generation, R8Scan synthesizes tests by injecting prioritized real-world functions, thereby introducing richer structural features. Third, R8Scan leverages LLMs to construct argument expressions for the injected functions, enabling broader exploration of their semantics. More importantly, unlike prior compiler fuzzing approaches that are mainly heuristic-driven, R8Scan is explicitly guided by empirical findings obtained from a large-scale study of historical R8 bugs.

**LLM-based fuzzing.** Recent studies have explored the use of large language models to assist software testing and bug detection in different ways [61, 66, 67]. Fuzz4All [61] uses LLMs together with dynamic autoprompting to generate diverse edge-case test programs for compilers across multiple programming languages. WhiteFox [66] targets deep learning compilers and employs a multi-agent LLM framework to synthesize test programs guided by white-box optimization analysis. Beyond test generation, KNighter [67] uses LLMs to synthesize static-analysis checkers for detecting kernel bugs. Other studies use neural models for probabilistic structure generation or iterative feedback to repair syntactic and semantic errors in generated programs [12, 46]. Our work differs from these studies in both goal and usage of LLMs. Rather than relying on LLMs to generate complete test programs or analysis rules end to end, R8Scan uses LLMs in a constrained role to construct argument expression sets for prioritized real-world functions. This design helps activate bug-relevant execution paths after function injection, while avoiding over-reliance on unconstrained model generation.

## 8 CONCLUSION

R8 is the default infrastructure for the Android build process. Despite the dominant role of R8, no prior research has investigated its bugs. To fill this gap, we conduct an investigation of R8 bugs. By analyzing 945 reported bugs, we find that the *Optimization*, *D8*, and *Shrinking* components of R8 are error-prone. Finally, we develop R8Scan, the first testing framework for R8. R8Scan synthesizes test cases by injecting prioritized functions derived from real-world code into seeds empowered by LLM-aided semantic exploration. R8Scan detects 17 R8 bugs. It also

detects 10 bugs in OpenJDK and 6 bugs in ART. We believe this study offers valuable insights into the security of the R8 compiler.

## Acknowledgments

We thank all reviewers for their constructive comments, and the editors for their kind arrangement. This research is supported by the National Natural Science Foundation of China (No.62372193, No.U2436207 and No.624B2067), and the Major Technological Innovation Program of HUST (No. 2025ZDKJCX05).

## References

[1] 2024. jacoco. https://github.com/jacoco/jacoco. Accessed: 2025-11.
[2] 2024. JavaFuzz. https://github.com/AzulSystems/JavaFuzzer. Accessed: 2025-11.
[3] Yameng Bai, Xingming Sun, Guang Sun, Xiaohong Deng, and Xiaoming Zhou. 2008. Dynamic k-gram based software birthmark. In *19th Australian Conference on Software Engineering (aswec 2008)*. IEEE, 644–649.
[4] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. JIT-Picking: Differential Fuzzing of JavaScript Engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 351–364. doi:10.1145/3548606.3560624
[5] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler bug isolation via effective witness test program generation. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 223–234. doi:10.1145/3338906.3338957
[6] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. 2023. Toward Understanding Deep Learning Framework Bugs. *ACM Trans. Softw. Eng. Methodol.* 32, 6 (2023), 135:1–135:31. doi:10.1145/3587155
[7] Junjie Chen, Haoyang Ma, and Lingming Zhang. 2020. Enhanced Compiler Bug Isolation via Memoized Search. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 78–89. doi:10.1145/3324884.3416570
[8] Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2025. Reasoning Runtime Behavior of a Program with LLM: How Far Are We? . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1869–1881. doi:10.1109/ICSE55347.2025.00012
[9] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2021. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1 (2021), 4:1–4:36. doi:10.1145/3363562
[10] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1257–1268.
[11] CPython-3.13. 2025. https://tonybaloney.github.io/posts/python-gets-a-jit.html. Accessed: 2025-11.
[12] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 95–105. https://doi.org/10.1145/3213846.3213848
[13] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, and Wangding Zeng. 2024. DeepSeek-V3 Technical Report. *CoRR* abs/2412.19437 (2024). arXiv:2412.19437 doi:10.48550/ARXIV.2412.19437
[14] The Rust Project Developers. 2025. The Rust Compiler (rustc). https://doc.rust-lang.org/rustc/ Accessed: 2025-01-09.
[15] Will Dietz, Peng Li, John Regehr, and Vikram S. Adve. 2012. Understanding integer overflow in C/C++. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 760–770. doi:10.1109/ICSE.2012.6227142
[16] Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. 2020. Closer to the Edge: Testing Compilers More Thoroughly by Being Less Conservative About Undefined Behaviour. In *Proceedings of the 35th IEEE/ACM International Conference on Automated*

*Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 1219–1223. doi:10.1145/3324884.3418933
[17] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Qi Alfred Chen. 2020. A comprehensive study of autonomous vehicle bugs. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 385–396. doi:10.1145/3377811.3380397
[18] GCC. 2025. https://gcc.gnu.org/. Accessed: 2025-11.
[19] Google. 2024. Android Open Source Project. https://source.android.com/. Accessed: 2025-11.
[20] Qi Guo, Xiaofei Xie, Shangqing Liu, Ming Hu, Xiaohong Li, and Lei Bu. 2025. Intention is All You Need: Refining Your Code from Your Intention . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1127–1139. doi:10.1109/ICSE55347.2025.00191
[21] The LLVM Compiler Infrastructure. 2024. LLVM: A Compilation Framework. https://llvm.org/. Accessed: 2025-11.
[22] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 510–520. doi:10.1145/3338906.3338955
[23] Issue-130135768. 2025. https://issuetracker.google.com/issues/130135768. Accessed: 2025-11.
[24] Issue-133686361. 2025. https://issuetracker.google.com/issues/133686361. Accessed: 2025-11.
[25] Issue-147411673. 2025. https://issuetracker.google.com/issues/147411673. Accessed: 2025-11.
[26] Issue-150688800. 2025. https://issuetracker.google.com/issues/150688800. Accessed: 2025-11.
[27] Issue-232379893. 2025. https://issuetracker.google.com/issues/232379893. Accessed: 2025-11.
[28] Issue-235733922. 2025. https://issuetracker.google.com/issues/235733922. Accessed: 2025-11.
[29] Issue-236691999. 2025. https://issuetracker.google.com/issues/236691999. Accessed: 2025-11.
[30] Issue-96076704. 2025. https://issuetracker.google.com/issues/96076704. Accessed: 2025-11.
[31] Haoxiang Jia, Ming Wen, Zifan Xie, Xiaochen Guo, Rongxin Wu, Maolin Sun, Kang Chen, and Hai Jin. 2023. Detecting JVM JIT Compiler Bugs via Exploring Two-Dimensional Input Spaces. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 43–55. doi:10.1109/ICSE48619.2023.00016
[32] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 96–105. doi:10.1109/ICSE.2007.30
[33] Zongze Jiang, Ming Wen, Jialun Cao, Xuanhua Shi, and Hai Jin. 2024. Towards Understanding the Effectiveness of Large Language Models on Directed Test Input Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 1408–1420. doi:10.1145/3691620.3695513
[34] Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. 2023. 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*, Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (Eds.). ACM, 66–79. doi:10.1145/3600006.3613140
[35] Shaohua Li and Zhendong Su. 2023. Finding Unstable Code via Compiler-Driven Differential Testing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 238–251. doi:10.1145/3582016.3582053
[36] Fang Liu, Ge Li, Qianhui Zhao, and Li Zhang. 2025. Learning to represent code semantics. *SCIENCE CHINA Information Sciences* 68, 7 (2025). doi:10.1007/s11432-023-3898-5
[37] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 196:1–196:25. doi:10.1145/3428264
[38] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. *Proceedings of the ACM Program. Lang.* 7, PLDI (2023), 1826–1847. doi:10.1145/3591295
[39] Google LLC. 2025. V8 JavaScript Engine. https://v8.dev/ Accessed: 2025-01-09.
[40] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 65–79. doi:10.1145/3453483.3454030
[41] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, Susan J. Eggers and James R. Larus (Eds.). ACM, 329–339. doi:10.1145/1346281.1346323
[42] Yifei Lu, Weidong Hou, Minxue Pan, Xuandong Li, and Zhendong Su. 2024. Understanding and Finding Java Decompiler Bugs. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 143 (April 2024), 27 pages. doi:10.1145/3649860

[43] Haoyang Ma, Wuqi Zhang, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. 2024. Towards Understanding the Bugs in Solidity Compiler. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 1312–1324. doi:10.1145/3650212.3680362

[44] OpenJDK. 2025. https://github.com/openjdk/jdk. Accessed: 2025-11.

[45] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015), 1155–1179. doi:10.1002/spe.2346

[46] Ravin Ravi, Dylan Bradshaw, Stefano Ruberto, Gunel Jahangirova, and Valerio Terragni. 2025. LLMLOOP: Improving LLM-Generated Code and Tests Through Automated Iterative Feedback Loops. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2025, Auckland, New Zealand, September 7-12, 2025*. IEEE, 930–934. https://doi.org/10.1109/ICSME64153.2025.00109

[47] Carolyn B. Seaman. 1999. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Trans. Software Eng.* 25, 4 (1999), 557–572. doi:10.1109/32.799955

[48] Mayank Sharma, Pingshi Yu, and Alastair F. Donaldson. 2023. RustSmith: Random Differential Compiler Testing for Rust. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1483–1486. doi:10.1145/3597926.3604919

[49] "Statista". 2025. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. Accessed: 2025-11.

[50] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 294–305. doi:10.1145/2931037.2931074

[51] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 361–371. doi:10.1145/3180155.3180236

[52] Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. 2022. Understanding and exploiting optimal function inlining. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch (Eds.). ACM, 977–989. doi:10.1145/3503222.3507744

[53] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch (Eds.). ACM, 697–709. doi:10.1145/3503222.3507764

[54] Yongqiang Tian, Xueyan Zhang, Yiwen Dong, Zhenyang Xu, Mengxiao Zhang, Yu Jiang, Shing-Chi Cheung, and Chengnian Sun. 2024. On the Caching Schemes to Speed Up Program Reduction. *ACM Trans. Softw. Eng. Methodol.* 33, 1 (2024), 17:1–17:30. doi:10.1145/3617172

[55] D8 Issue Tracker. 2025. https://issuetracker.google.com/issues?q=componentid:317603. Accessed: 2025-11.

[56] R8 Issue Tracker. 2025. https://issuetracker.google.com/issues/new?component=326788. Accessed: 2025-11.

[57] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. 2023. FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler. In *Proceedings of the 32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 1865–1882. https://www.usenix.org/conference/usenixsecurity23/presentation/wang-junjie

[58] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 326–337. doi:10.1145/3338906.3338962

[59] Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. 2023. 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*. IEEE, 56–68. doi:10.1109/ICSE48619.2023.00017

[60] Mingyuan Wu, Yicheng Ouyang, Minghai Lu, Junjie Chen, Yingquan Zhao, Heming Cui, Guowei Yang, and Yuqun Zhang. 2023. Sjfuzz: Seed and mutator scheduling for jvm fuzzing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1062–1074.

[61] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[62] Zifan Xie, Ming Wen, Tinghan Li, Yiding Zhu, Qinsheng Hou, and Hai Jin. 2024. How Does Code Optimization Impact Third-party Library Detection for Android Applications?. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1919–1931.

[63] Zifan Xie, Ming Wen, Shiyu Qiu, and Hai Jin. 2025. Validating JVM Compilers via Maximizing Optimization Interactions. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*

(Hilton La Jolla Torrey Pines, La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 345–360. doi:10.1145/3622781.3674188

[64] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. 2023. Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 3655–3672. https://www.usenix.org/conference/usenixsecurity23/presentation/xu-jianhao

[65] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Gaosen Zhao, Yu Jiang, and Chengnian Sun. 2023. Pushing the Limit of 1-Minimality of Language-Agnostic Program Reduction. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 636–664. doi:10.1145/3586049

[66] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. Whitefox: White-box compiler fuzzing empowered by large language models. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 709–735.

[67] Chenyuan Yang, Zijie Zhao, Zichen Xie, Haoyu Li, and Lingming Zhang. 2025. KNighter: Transforming Static Analysis with LLM-Synthesized Checkers. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP 2025*. ACM, 655–669.

[68] Geunha You, Gyoosik Kim, Seong-je Cho, and Hyoil Han. 2021. A Comparative Study on Optimization, Obfuscation, and Deobfuscation tools in Android. *J. Internet Serv. Inf. Secur.* 11, 1 (2021), 2–15. doi:10.22667/JISIS.2021.02.28.002

[69] Mengxiao Zhang, Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Shin Hwei Tan, and Chengnian Sun. 2024. LPR: Large Language Models-Aided Program Reduction. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 261–273. doi:10.1145/3650212.3652126

[70] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 129–140. doi:10.1145/3213846.3213866

[71] Yingquan Zhao, Zan Wang, Junjie Chen, Ruifeng Fu, Yanzhou Lu, Tianchang Gao, and Haojie Ye. 2024. 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*, Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie (Eds.). ACM, 3943–3957. doi:10.1145/3658644.3690366

[72] Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. 2021. An empirical study of optimization bugs in GCC and LLVM. *Journal of Systems and Software* 174 (2021), 110884. doi:10.1016/J.JSS.2020.110884

[73] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *Comput. Surveys* 54, 11s (2022), 230:1–230:36. doi:10.1145/3512345