Validating JVM Compilers via Maximizing Optimization Interactions

Zifan Xie[*][†]
Huazhong University of Science and Technology, Wuhan, China
xzff@hust.edu.cn

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

Shiyu Qiu[*][†]
Huazhong University of Science and Technology, Wuhan, China
m202372059@hust.edu.cn

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

Abstract

This paper introduces the concept of optimization interaction, which refers to the practice in modern compilers where multiple optimization phases, such as *inlining*, *loop unrolling*, and *dead code elimination*, are not completed in a one-off sequential order while being interacted instead. Therefore, while optimizing a certain phase, the compiler needs to ensure that the results of other optimization phases will not be disrupted, as this could lead to compiler crashes or unpredictable results. To verify whether compilers can correctly handle the optimization process across various phases, we propose MopFuzzer, which aims at maximizing runtime optimization interactions during fuzzing. Specifically, it encourages the JVM to perform multi-stage optimizations and verifies the correctness of the compiler's optimized code through differential testing. Currently, MopFuzzer has implemented 13 mutators, and each is intended to trigger a certain optimization behavior. Such mutators are applied iteratively to the same program point, aiming to maximize optimization interactions. Subsequently, the testing process is guided by a novel method based on profile data, which records the optimization behaviors performed by the compiler. The guidance enables MopFuzzer to generate mutants that are able to maximize optimization behaviors and their interactions. Our evaluation has led to 59 bug reports for widely used production JVMs, OpenJDK and OpenJ9.

Keywords: JVM, JIT Compiler, Optimization

ACM Reference Format:
Zifan Xie, Ming Wen, Shiyu Qiu, and Hai Jin. 2024. Validating JVM Compilers via Maximizing Optimization Interactions. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3622781.3674188

1 Introduction

Compiler optimization plays a pivotal role in modern programming language virtual machines, significantly enhancing application performance. For example, the Java Virtual Machine (JVM) employs advanced optimization techniques to translate frequently executed bytecode into machine code, thereby boosting performance. This process is facilitated by the Just-In-Time (JIT) compiler, an integral and indispensable component within various JVM implementations (e.g., HotSpot in OpenJDK) as well as in virtual machines for other programming languages (e.g., JavaScript V8 Turbofan [47]).

The JVM's JIT compiler has implemented a wide array of intricate optimization techniques, including *lock elimination*, *loop unrolling*, and *function inlining*. Recognized as one of the most significant yet complex components in JVM, the JIT compiler has witnessed an increase in the occurrence of bugs, becoming more pervasive over recent years [23]. Unfortunately, ensuring the accuracy of JIT compilers is challenging. Despite the commendable performance achieved by many existing testing tools [6, 7, 23, 28, 53], a recent study revealed that the majority of disclosed bugs are shallow [28], which are often triggered at the initial parsing and verification stages. Besides, we observe that various optimization

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

```
// src/hotspot/share/c1/c1_GraphBuilder.cpp
bool GraphBuilder::try_inline_full(...){
  // ...
  if (callee->is_synchronized() &&
      sync_handler->state() != NULL) {
    // If an exception is thrown and not
    // handled within an inlined
    // synchronized method, the monitor must
    // be released before the exception is
    // rethrown in the outer scope.
    fill_sync_handler(lock, sync_handler);
} }
```

Listing 1. An example of OpenJDK that carefully handles the optimization interaction during function inlining

behaviors frequently *interact* with each other, further complicating the validation of compiling correctness.

Optimization Interaction. The JVM JIT compiler typically optimizes code through a series of phases. For instance, in the C2 phase, the compiler performs optimizations such as *iterative global value numbering*, *inlining*, and *autobox elimination* [13, 18, 33]. Although each phase has its designated task, the complex code implemented by developers in practice can often lead to intricacies where multiple optimizations are intertwined. In this work, we call such interaction across multiple optimization phases as *optimization interaction*. Listing 1 shows an example where OpenJDK carefully handles optimization interaction during the function inlining phase. In this example, when attempting to inline a callee during function inlining, the compiler enforces specific exception handling to tackle synchronized methods. This includes instructions to release the monitor lock and re-throw the exception. Such handling is crucial to ensure the correctness and consistency of a program's synchronization behavior. In this way, the JVM can maintain the thread safety of synchronized methods, even in the context of optimized inline calls. Without such correct handling, it could lead to issues like *monitor lock leakage* and *improper exception handling* in subsequent lock-related phases.

Key Idea. Despite developers' substantial efforts to safeguard optimizations in one phase from negatively impacting the correctness of other phases, existing compilers still remain susceptible to various bugs caused by intricate scenarios in practice. Unfortunately, effective testing is challenging as the triggering of such optimization interactions often requires sophisticated test cases. This study introduces an innovative method aimed at identifying deep JIT compiler bugs stemming from optimization interactions instead of those shallow ones that can be triggered easily at the early compilation stages. To effectively and efficiently achieve this goal, we need to address several non-trivial challenges. First, the generated test seeds should proactively induce optimization behaviors, particularly the interactions among diverse optimization strategies. Second, the search space will be huge considering the interacting impact of substantial optimizations, and thus effective guidance is desired to navigate the exploration of optimization interactions.

Our key idea to tackle the aforementioned challenges primarily encompasses the following aspects. First, we devise a series of novel *optimization evoking* mutators intended to proactively trigger different types of optimization behaviors to generate test seeds. Specifically, we employ these mutators on a fixed mutation point iteratively, and the inserted new code is adjacent to or nested around existing code. This strategy aims to maximize the occurrence of diverse optimization interactions. We then probe the *profile data* generated by virtual machines during runtime and leverage the contained optimization information as valuable feedback, which is used to navigate the generation of new seeds. Ultimately, any crashes or inconsistencies observed across multiple JVM implementations serve as test oracles.

Implementation. We develop MopFuzzer based on our idea to test JVM JIT compilers and validate their correctness. Specifically, we design and integrate a broad range of optimization evoking mutators that can proactively trigger different types of optimizations. By leveraging the profile data extracted from the log information generated by JVM as guidance, MopFuzzer is able to examine whether a generated mutant can indeed trigger more optimization behaviors. Eventually, MopFuzzer has uncovered 59 bugs, of which 45 are in OpenJDK and 14 in OpenJ9. We find that all the test cases triggering these bugs involve various JVM optimization behaviors. Such results validate our observation, indicating that the mishandling of optimization interactions among diverse optimization techniques ultimately led to the bugs. Our results also show that MopFuzzer can outperform the state-of-the-art approaches in terms of code coverage and the increment of triggered optimization behaviors. Section 4 presents our evaluation and detailed analysis.

Contributions. Our main contributions are as follows:

• We are the first to focus on exploring the optimization interactions of JIT compilers.

• We propose novel ideas with optimization evoking mutators and profile data-based guidance to rigorously test JVM JIT compilers via maximizing optimization interactions.

• We substantiate our idea as an automated testing tool MopFuzzer, which has detected 59 bugs in widely used JVM implementations, OpenJDK and OpenJ9.

Our tool MopFuzzer and all the reported bugs are publicly available to facilitate future researches.

https://github.com/CGCL-codes/MopFuzzer.

2 Background and Motivation

2.1 JVM and JIT Compiler

Initially, the JVM executes code (`.class` file) in the interpretive mode. During this process, it profiles runtime data, identifying frequently executed functions or basic blocks, which are then labeled as "HotSpot" code. To enhance the efficiency of executing such HotSpot code, the JVM employs sophisticated optimization strategies, which translate them into machine code with high-performance. The compiler facilitating this transformation is known as the *Just-In-Time* (JIT) compiler [3, 41]. The JIT compiler is an integral and crucial component of many JVM implementations (e.g., HotSpot in OpenJDK). The JVM JIT mainly contains two compiler types: *C1 and C2 Compiler.* The *C1 Compiler* [25], or "Client Compiler", prioritizes swift start-up times, making it historically ideal for short-lived applications. *C2 Compiler* [33], known as the "Server Compiler", is designed for optimal long-term performance, traditionally suiting long-running server-side applications.

2.2 JVM Flags and Profile Data

The JVM also provides abundant flags [14, 15] to print out the profile data to monitor JVM's runtime behaviors for debugging and analysis. In this study, we refer to profile data as the log information that records the details of certain optimization behaviors performed by the JVM. For instance, the *PrintAssembly* flag can be used to output the assembly code generated by the JIT compiler. The *PrintEscapeAnalysis* flag outputs the details of how the JIT compiler is optimizing object allocations based on the escape analysis. This is especially useful for those who want to understand the low-level optimizations performed by the JVM on the Java bytecode.

2.3 Optimization Interaction

Code optimization is a crucial part of the whole compilation process for modern compilers [10, 11, 19, 34, 39]. Take LLVM [27] as an example. It breaks down the optimization process into a series of "passes", with each pass responsible for a specific task. Similarly, the JVM JIT compiler also optimizes code through a series of phases. In the C2 phase, the compiler performs optimizations like *iterative global value numbering*, *inlining*, *autobox elimination*, etc. Although each phase has its designated task, the code implemented by developers with unlimited complexity in practice often leads to intricacies where multiple optimizations become intertwined. Therefore, developers need to make tremendous efforts to ensure that optimizations in one phase do not adversely affect the correctness of other phases. In this work, we call such interaction of optimizations across multiple phases as ***optimization interaction***.

Listing 1 shows an example where OpenJDK carefully handles optimization interaction during the function inlining phase. In this example, when attempting to inline a callee during the function inlining phase, the compiler generates specific exception handling code for synchronized methods. This includes instructions to release the monitor lock and rethrow the exception. Such handling is crucial to ensure the correctness and consistency of a program's synchronized behavior. In this way, the JVM maintains the thread safety of synchronized methods, even in the context of optimized inline calls. On the other hand, if the compiler does not carefully handle synchronized methods during the inlining phase, it could lead to issues like *monitor lock leakage* and *improper exception handling* in subsequent lock-related phases.

We observe that the phenomenon of optimization interaction is pervasive in modern compilers while guaranteeing its correctness is challenging since such interaction can only be triggered with sophisticated test cases. In particular, multiple optimization behaviors should be triggered simultaneously (e.g., a callee is involved in both synchronization and inline optimizations for the example in Listing 1). Generating such test cases is non-trivial, and the following shows a real example demonstrating how our designed approach, MopFuzzer, can achieve such a goal effectively.

2.4 Motivating Example

Listing 2 shows a seed collected from existing regression tests. We show how MopFuzzer can generate a new seed based on it that can trigger a crash in OpenJDK via exploring optimization interactions. First, MopFuzzer randomly selects one statement to serve as the *Mutation Point* (MP). It then iteratively applies various mutators to MP, each designed to evoke a specific type of JIT optimization behavior. Listing 3 illustrates the mutant generated after four iterations, where MopFuzzer applied four mutators to MP: *LockElimination-evoke* twice for the JIT lock elimination optimization, *LoopUnroll-evoke* once for loop unrolling, and *LockCoarsening-evoke* once for lock coarsening optimization. Eventually, the generated mutant led the compiler to crash, and we reported this bug to the OpenJDK developers. It was confirmed as JDK-8312744. The whole process aims to generate a mutant that can trigger complex JIT optimization interactions, and thus MopFuzzer mutates the seed regarding the same MP, which results in the code inserted by MopFuzzer being nested or adjacent. For example, the two synchronized statements at Line 8 and Line 6 are nested, while the synchronized statement at Line 13 resides next to the *for-loop* structure at Line 10. As a result, when the compiler optimizes the mutant, different optimization behaviors (e.g., *lock elimination*, *loop unrolling*, and *lock coarsening* in this case) are likely to interact with each other.

```
1 class T {
2   public static void main(String[] arg){
3     T t = new T();
4     for (int i = 0; i < 50_000; ++i) {
5       t.foo(i);  // Mutation Point
6     }
7   }
8   void foo(int i) { ... }
9 }
```

Listing 2. The original seed with line 5 selected as the *mutation point*

```
1  class T {
2    public static void main(String[] arg){
3      T t = new T();
4      for (int i = 0; i < 50_000; ++i) {
5        // iter1: LockElimination-evoke
6        synchronized (T.class) {
7          // iter2: LockElimination-evoke
8          synchronized (T.class) {
9            // iter3: LoopUnroll-evoke
10           for (int v0 = 0; v0 < 8; ++v0)
11             t.foo(i);
12         // iter4: LockCoarsening-evoke
13         }
14         synchronized (T.class) {
15           t.foo(i); // Mutation Point
16         }
17       }
18     }
19   }
20   void foo(int i) { ... }
21 }
```

Listing 3. A test case that causes a crash (i.e., JDK-8312744 [21]) in the mainline OpenJDK. After four iterations, MopFuzzer applied four mutators at the *mutation point*. The code inserted in each iteration is highlighted for clarity. Additionally, the test case has been condensed for brevity.

We observe that the incorrect handling of such interaction among diverse optimizations that eventually caused the crash. In particular, applying a single mutator among the four to the original seed or removing any injected code by any mutator from the mutant cannot cause the compiler to crash anymore. In this case, the root cause of the error is the interaction between *loop unrolling optimization* and *lock coarsening optimization*. The loop unrolling optimization in the ideal loop phase unfolds the loop structure at Lines 10-11, altering the program structures. This alteration, in turn, results in a failure in the lock coarsening optimization during the macro expand phase. Then, the compiler attempts to retry without lock coarsening, but it mistakenly sets a pointer to *null* and uses it in subsequent optimizations, thus resulting in a *null* pointer reference error.

Although developers have recognized the importance of handling the interplay among various optimization behaviors to prevent compilers from inadvertently disrupting the logic of another optimization behavior (see Listing 1), they can still easily overlook certain situations that involve complex optimization interactions, and thus bugs might occur. However, guaranteeing the correctness of the JIT compiler is of significant importance. Therefore, it motivates us to propose a method that can proactively identify such bugs via *rigorously* testing compilers with the aim of maximizing optimization interactions.

2.5 Existing Practices

A number of JVM fuzzing tools have been proposed over recent years [6, 7, 23, 52, 53]. In particular, *JITFuzz* [50] and *Artemis* [28] are two state-of-the-art JVM testing tools that are proposed recently. JITFuzz employs four mutators tailored to trigger specific JVM optimization behaviors: function inlining, simplification, scalar replacement, and escape analysis. Additionally, it has designed two mutators that reshape the control flow of seeds. JITFuzz then adopts a coverage-driven strategy, randomly picking mutation points with such mutators in each iteration to exploit JVM bugs. Artemis introduced the concept of Compilation Space Exploration for the first time [28]. Specifically, it employs three mutation templates that target method calls, loops, and uncommon traps respectively, thus manipulating the dynamic behaviors between interpretation and JIT compilation. While these tools have proven to be effective, they still fall short in detecting bugs related to optimization interactions for the following reasons. First, triggering such JIT bugs requires the JVM to activate diverse optimization behaviors in one execution, such as reflective mechanisms, nested locks, and complex loop structures. For instance, JDK-8312744 requires a test case to involve nested locks and loop structures. The mutators designed by JITFuzz and Artemis overlook the interaction among various optimization behaviors. Since the mutants generated by them lack complex structures (e.g., nested and adjacent locks as shown in Listing 3), they fail to explore the interactions among different JVM optimization behaviors, and thus cannot validate their correctness.

3 Approach

In this work, we introduce MopFuzzer to test JVM JIT compilers, which validate compiler behaviors via maximizing the interactions among diverse optimization behaviors. To achieve such a goal, we first design a variety of mutators, each of which intends to proactively evoke certain optimization behaviors in the target compiler. For example, the mutator *LoopUnroll-evoke* inserts a carefully designed *loop structure* into the seed, aiming to trigger loop unrolling optimizations in the compiler. MopFuzzer then applies multiple mutators to the same code element of a seed iteratively guided by the profile data, aiming to maximize optimization interactions. Specifically, the code inserted by the mutators is designed to be *adjacent to* or *nested around* the selected mutation points to explore optimization interactions. Correctly handling such seeds requires the compiler to ensure that implementing one optimization does not compromise the results of another, and thus can validate the compiler's stability.

3.1 Overview

Algorithm 1 shows the general workflow of our approach. Specifically, it takes three inputs: a seed corpus C, a target JVM implementation $\mathcal{J}$, and the designed mutators $\mathcal{M}$. MopFuzzer mainly contains the following steps.

1. Select a seed file p as parent from C sequentially (Line 1); Randomly select a statement MP as the mutation point in p (Line 2); Execute p to obtain the corresponding profile data O_p (Line 3), the profile data records the various optimization behaviors performed by $\mathcal{J}$; Initiate the weight of each mutator to 1 (Line 4).
2. Analyze the code structures of MP, identify its optimization space, select a set of mutators $\mathcal{M}_p$ that are applicable to MP, and obtain the corresponding weights (i.e., $\mathcal{W}_p$) for these mutators (Lines 6-7).
3. Choose a mutator m from $\mathcal{M}_p$ by weighted random selection (Lines 8-9).
4. Apply mutator m on MP to obtain a child mutant c and further execute the mutant c to obtain its corresponding profile data O_c (Lines 10-13).
5. Compare O_c with O to identify the increment of optimization behaviors and update the mutator m's weight W_m accordingly (lines 13-14).
6. Set mutant c as the parent and repeat steps 2-5 until a predetermined threshold of iteration times is reached, or a mutant causes the JVM $\mathcal{J}$ to crash (Line 18).
7. Perform differential testing on the final mutant $c^\star$ using different JVM implementations. If MopFuzzer identifies different outputs, a potential bug might lurk within them. We will reduce and report the mutant if $c^\star$ indeed triggers bugs. (Lines 19-22).

Algorithm 1: MopFuzzer's Fuzzing Loop

```
Input :A seed corpus $C$; a target JVM implementation $\mathcal{J}$. The
       designed mutators $\mathcal{M}$.
1  $p \leftarrow getSeed(C)$ // Obtain a seed file as parent from $C$
2  $MP \leftarrow selectMP(p)$ // Select a mutation point on $c$
3  $O_p \leftarrow execute(\mathcal{J}, p)$ // Execute $p$ to get profile data $O_p$
4  $\mathcal{W} \leftarrow \{1, 1, ..., 1\}$ // Initial mutators' weight
5  repeat
6      /* find applicable mutators and their weights   */
7      $\mathcal{M}_p, \mathcal{W}_p \leftarrow applicableMutators(MP)$
8      /* select mutator $m$ based on its weight        */
9      $m \leftarrow selectMutatorByWeight(\mathcal{M}_p, \mathcal{W}_p)$
10     /* Apply mutator $m$ on $MP$ to get a child mutant */
11     $c \leftarrow applyMutator(MP, m)$
12     /* Execute the mutant $c$ to get profile data $O_c$  */
13     $O_c \leftarrow execute(c, \mathcal{J})$
14     /* Update the mutator $m$'s weight $W_m$ based on
          optimization behaviors increment by compare
          profile data between $O_p$ and $O_c$            */
15     $W_m \leftarrow UpdateWeight(O_p, O_c)$
16     /* Set the parent as the child mutant $c$          */
17     $\langle p, O \rangle \leftarrow \langle c, O_c \rangle$
18 until MAX Iterations or Crash found;
19 /* Conduct differential testing on final mutant $c^\star$  */
20 $differential(c^\star)$
21 /* Reduce and report mutant if it trigger bugs      */
22 $reduceAndReport(c^\star)$
```

3.2 Optimization Evoking Mutators

As aforementioned, to maximize optimization interactions, we first need to design a set of mutators that can proactively trigger specific optimization behaviors of JVM JIT compilers. Different JVM implementations, such as HotSpot, OpenJ9, and GraalVM, might use varying optimization techniques and strategies. Therefore, designing such optimization evoking mutators is non-trivial.

In this work, we refer to the optimization strategies employed by HotSpot. HotSpot, utilized in OpenJDK and Oracle JDK, is known for its robust performance and remains to be one of the most popular and widely used JVM implementations nowadays. On the official OpenJDK Wiki [49], the HotSpot team has listed a comprehensive set of optimization techniques used in its JIT compiler. Certain optimization strategies involve deep-level optimizations, making them difficult to be activated via mutations. For example, the "Local Code Scheduling" aims to improve the execution efficiency of program on specific hardware by rearranging the sequence of scheduled instructions and hence designing mutators at the source code level is non-trivial. Eventually, we designed 13 Optimization Evoking Mutators, which aim to trigger primarily the following optimization behaviors respectively: loop unrolling, lock elimination, lock coarsening, inlining, dereflection, loop peeling, loop unswitching, deoptimization, autobox elimination, redundant store elimination, algebraic simplification, escape analysis, dead code elimination. Due to page limit, we list five of the designed mutators in Table 1, and other mutators' details can be found on our project page [35]. We select these 13 optimizations since they are the most common ones that are pervasively triggered. Note that due to the interactive nature of various optimization behaviors, one mutator can actually trigger multiple types of optimization behaviors in practice (see more discussion in Section 3.4). Besides, there are multiple ways to design the evoking mutator for each optimization behavior, and we only explored one implementation in this study. Our idea is extensible, and thus the support for more optimizations as well as the other implementations of such evoking mutators are left as our important future work.

LoopUnrolling-evoke: Loop unrolling is a common optimization technique used in programming to enhance the

Table 1. Five optimization evoking mutators (5/13) designed in MopFuzzer. We use the statement $m = a + t.f()$ as the Mutation Point (MP). The "Cond" column indicates whether applying this mutator requires MP to contain certain code elements. The examples show how the mutators insert or change MP. The updated MP (i.e., MP_n) will be used for subsequent iterations.

Mutator	Illustration	Cond	Example
LoopUnrolling-evoke	Insert a loop structure before MP. The loop structure wraps a copy of MP. We do not use the copy of MP as MP_n for performance considerations.	✗	$+\ for\ (int\ i = 0; i < N; i{+}{+})$ $+\quad m = a + t.f();$ $\quad m = a + t.f();$ // MP_n
LockElimination-evoke	Wrap MP in a *synchronized* body. The synchronized object can be any valid object or the class constant.	✗	$+\ synchronized\ (T.class)\ \{$ $\quad m = a + t.f();$ // MP_n $+\ \}$
LockCoarsening-evoke	If MP is in a *synchronized* body, we split this body into two synchronized bodies with the same synchronized object.	✓	$synchronized\ (T.class)\ \{$ ... $+\ \}\ synchronized\ (T.class)\ \{$ $\quad m = a + t.f();$ // MP_n $\}$
Inlining-evoke	If MP contains a binary expression, we replace it with a function call, with the variables involved in the binary expression passed as arguments to the function.	✓	$-\ m = a + t.f();$ $+\ m\ =\ foo(a,\ t.f());$ // MP_n ... $+\ int\ foo(int\ x,\ int\ y)\ \{\ return\ x + y;\ \}$
DeReflection-evoke	If MP contains a function call or field access, we replace the function call or field access with reflection call through the Java *reflection mechanism.*	✓	$-\ m = a + t.f();$ $+\ Class\langle ?\rangle\ CT\ =\ Class.forName(\text{“}T\text{”});$ $+\ m = a + CT.getDeclaredMethod(\text{“}f\text{”}).invoke(t);$ // MP_n

performance of loops. Specifically, it involves replicating the loop body multiple times to reduce the number of iterations, thus decreasing the overhead associated with the loop control. To proactively trigger this optimization, we design mutator *LoopUnrolling-evoke*, which inserts a loop structure that wraps a copy of the MP before the original MP. We do not use the copy of MP as MP_n (i.e., the MP used for subsequent iterations) for performance considerations (to avoid nested loop structures). Specifically, if we do not introduce a copied version of MP, it can be easily wrapped by nested loops if MopFuzzer chooses this mutator multiple times. Such nested loop structures can significantly enhance the overhead of executing this program, thus degrading testing efficiency. Applying this mutator requires no condition.

LockElimination-evoke: Lock elimination optimization is useful in contexts where multithreading is involved. Specifically, it focuses on removing unnecessary locking and unlocking operations, and thus enhances the performance of program by reducing resources spent on lock management. The *LockElimination-evoke* mutator aims to evoke the lock elimination optimization proactively in compilers. In particular, it wraps the MP in a synchronized body and the synchronized object can be a valid object (e.g., '`this`' pointer) or a class constant. Applying this mutator requires no condition.

LockCoarsening-evoke: Lock coarsening is used to reduce the overhead of frequently acquiring and releasing locks in a program. If the same lock is repeatedly acquired and released in successive operations, coarsening the scope of the lock can reduce the number of lock operations. This optimization enables a lock to be held for a longer period while decreasing the total number of lock requests, thereby improving efficiency. The *LockCoarsening-evoke* mutator aims to trigger such lock coarsening optimization in compilers. Applying this mutator ***requires*** the MP to be enclosed in a synchronized body. MopFuzzer will split this body into two synchronized bodies with the same synchronized object.

Inlining-evoke: Function inlining involves replacing a function call with the actual code of the function itself. This means that whenever the program calls a function, the compiler directly inserts the entire code of that function at the call site instead of performing a regular function call. The primary benefit of this approach is to reduce the overhead of function calls, such as setting up the call stack and passing parameters. Most importantly, it can increase the optimization scope by optimizing the callee and caller collectively. The *Inlining-evoke* mutator ***requires*** a binary expression contained in MP and replaces the binary expression with a new function call, passing the two operands involved in the binary expression as arguments to the function. Additionally, MopFuzzer generates the corresponding declaration of this function, which performs the same operation as the original binary expression and passes the original operands as parameters. In our implementation, we support operands of expressions that are primitive data types.

DeReflection-evoke: Reflection elimination boosts the performance of code using Java's reflection APIs. Reflection, which enables dynamic object creation, method invocation, and field access at runtime, often slows down execution due to extra checks and indirect addressing. This optimization replaces slower reflection calls with direct calls when the compiler can identify specific methods or field types, thus enhancing code efficiency. The *Inlining-evoke* mutator ***requires*** a function call or field access in the MP, and replaces the function call or field access with a reflection call through the Java reflection mechanism.

3.3 Mutants Generation

After selecting a seed program p from the collected corpus C, MopFuzzer will first identify a set of mutators $\mathcal{M}_p$ that are applicable to MP based on their conditions. Among the designed 13 mutators, 6 types are unconditional and thus are all applicable to MP. For the remaining *conditional mutators*, MopFuzzer determines whether MP satisfies the corresponding condition. For example, for the statement of $m = a + b$, since the presence of a binary expression in MP satisfies the condition of Inlining-evoke (i.e., it requires a binary expression and transforms it into a function call), Inlining-evoke is added to $\mathcal{M}_p$. On the other hand, the absence of any function calls or field access in the AST means that DeReflection-evoke is not applicable to MP.

After obtaining $\mathcal{M}_p = \{m_1, m_2, ...m_n\}$, MopFuzzer gathers the corresponding weights for these mutators, which are denoted as $\mathcal{W}_p = \{w_1, w_2, ...w_n\}$ where n is the size of $\mathcal{M}_p$. Then, based on $\mathcal{W}_p$, MopFuzzer randomly selects a mutator from $\mathcal{M}_p$ and applies it to MP, thereby generating a child mutant c. Specifically, we select a mutator by the following potential function:

$$potential(m_i) = \frac{w_i}{\sum_{j=1}^{n} w_j} \tag{1}$$

Accordingly, the higher weight for a mutator, the higher probability for MopFuzzer to select it. Note that the weight of each mutator is initially set to 1 in the initialization phase (Lines 4 in Algorithm 1), and is dynamically updated during the iterative testing process (see Section 3.4).

3.4 Profile Data-based Guidance

The optimization interaction space is huge, and thus effective guidance is desired to maximize optimization interactions. In this study, we examine whether the generated mutants can actually trigger more optimization behaviors with respect to both distinct types and frequency. Specifically, we utilize JVM flags provided by JVM to print out the profile data since the output from such flags can reflect the compiler's optimization behaviors. To capture the optimization behaviors related to our designed mutators, we identify 15 flags specifically related to printing optimization behaviors via manual analysis, which are listed in our online website [35]. Such 15 flags can record 19 types of optimization behaviors. Via further reviewing the code repository of OpenJDK, we summarized the regular expression rules to capture the occurrences of each optimization behavior from the printed profile data.

```
// src/hotspot/share/opto/loopTransform.cpp
// Unroll the loop body one step
bool PhaseIdealLoop::do_unroll(...) {
  // ...
  if (TraceLoopOpts) {
    if (trip_count() < (uint)LoopUnrollLimit){
      tty->print("Unroll %d(%2d) ",
          unrolled_count()*2, trip_count());
    } else {
      tty->print("Unroll %d ", unrolled_count
          ()*2);
    } } }
```

Listing 4. Illustration of capturing the optimization behavior for loop unrolling using regular rules: `Unroll [0-9]+`

Take flag *TraceLoopOpts* as an example, it can print the information about optimization behaviors like loop unrolling and loop peeling. Particularly, Listing 4 shows code snippets from OpenJDK for loop unrolling optimization. When *TraceLoopOpts* is enabled, and if the JVM performs a loop unrolling optimization, it prints either "`Unroll %d(%2d)`" or "`Unroll %d`". Therefore, we can design a regular expression rule "`Unroll [0-9]+`" to capture the frequency of the loop unrolling optimization performed by JVM. We perform manual investigation for all the 15 selected flags and then design the regular expression rules accordingly, which are listed in our website [35]. To calculate the frequency of different optimization behaviors, MopFuzzer uses a 19-dimensional vector named *Optimization Behavior Vector* (OBV), where each dimension records the frequency of the corresponding optimization behavior. *Such a vector can reflect the interaction among diverse optimizations since it records both optimization types and frequencies.* Let us consider $OBV_p = \langle 1, 0, 0, 0, \dots, 0 \rangle$ for a parent and $OBV_c = \langle 2, 2, 2, 0, \dots, 0 \rangle$ for a child mutant, we can measure the increment between OBV_p and OBV_c to examine whether new optimization behaviors are triggered. Therefore, we introduce a metric, Δ, which is determined as the Euclidean distance between OBV_p and OBV_c, formally defined as:

$$\Delta = \sqrt{\sum_{i=1}^{n} (max(0, OBV_{ci} - OBV_{pi}))^2} \tag{2}$$

where OBV_{pi} and OBV_{ci} denote the ith elements of vectors OBV_p and OBV_c respectively. In this formula, $max(0, x)$ ensures that only increases are considered while those reductions will be ignored. The summation extends over all vector elements, capturing the combined increment across all monitored optimization types and their frequencies. In our example, $\Delta = 3$ is derived from the increment for the first three items of the vector.

The value of Δ reflects the increment of optimization behaviors interactions induced by applying a mutator. Thus, a higher Δ suggests a greater likelihood that applying this mutator will evoke the compiler in triggering different optimization behaviors *w.r.t.* types and frequency. Therefore, if a mutator results in a higher Δ , we opt to increase its weight w_m. This adjustment makes the mutator more likely to be chosen in the next round. Specifically, we update the weight of mutator m using the following formula:

$$w_m = w_m * (1 + \Delta/||OBV_c||) \tag{3}$$

where $||OBV_c||$ measures the magnitude of OBV_c and w_m is the weight of the mutator m. This formula strategically adjusts the weight of mutators that induce more optimization behaviors (i.e., a larger Δ value) during the fuzzing process, thus enabling them to be selected with higher probabilities in subsequent process.

Rationale Behind the Weighting Scheme. The weighting scheme explores complex interactions among various optimizations rather than merely increasing the total number of performed optimizations. By employing the Euclidean distance (Δ), we emphasize the changes across all types of optimizations, rewarding mutators that induce diverse behaviors. The alternative scheme of using the sum of optimization occurrences as weights was not adopted due to the occurrences of different types of optimization behaviors are imbalanced. For instance, after applying a mutator, certain behaviors (e.g., Inlining) might increase from 100 to 200 occurrences, while some only increase from 1 to 2 (e.g., LoopUnswitching) since they are harder to trigger. In this example, taking the sum (i.e., 202) as weight would be biased to mutators that triggering *Inlining* behaviors. Formula 3 can mitigate this issue. It calculates the Euclidean distance between the optimization behavior vectors of a parent and its child, and normalizes their distance by the magnitude of the child's OBV (i.e., $||OBV_c||$). The normalization adjusts weights to promote those mutators that not only trigger optimizations but also enhance the variety and frequency of optimization behaviors. Note that emphasizing frequency is also important, as some bugs can only be triggered when an optimization occurs multiple times. For instance, JDK-8324174 [22] exposes the bug through the use of three nested locks. Therefore, increasing the weight of mutators that have already been triggered is justified. Consequently, we do not simply increase the weight of mutators that have not yet been activated since this alternative will lose its effectiveness once each mutator has been selected at least once.

3.5 Bug Detection and Test case Reduction

MopFuzzer detects bugs by the following two test oracles:

Crash: A bug is detected if the target JVM crashes during runtime. The cause of the crash can be examined in an automatically generated file named `hs_err_pid.log`.

Miscompilation: After reaching the maximum number of iterations, we perform differential testing on the final mutant $c^\star$. Specifically, $c^\star$ is executed using different JVMs, including the Long-Time-Support (LTS) versions (i.e., 8, 11, 17, 21) and the mainline version (i.e., 23) of OpenJDK and OpenJ9. We then compare the consistency of the program's output. However, inconsistent results do not always signify real bugs, as they might stem from the use of random numbers, leading to unpredictable outcomes. Thus, we further carry out manual investigations to discern bugs and verify whether such inconsistencies are indeed caused by optimization errors.

Test Case Reduction. MopFuzzer often generates complex mutants by applying diverse mutators iteratively at the same program point. As a result, the test cases grow in size and trigger more optimization behaviors. However, the complexity prevents developers from easily understanding the root cause of the bugs. In fact, the root cause of these bugs could often be traced back to a few key structures. For example, triggering bug JDK-8324174 [22] requires a nesting of three layers of lock structures. To simplify this process, we adopt a semi-automated approach to reduce the size of the test case. Specifically, we manually strip away code inserted by MopFuzzer until the test case only involves key structures, ensuring that removing any further code would not trigger the bug. Lastly, we employ a leading test case reduction tool, ***perses*** [43], to further decrease the size of a test case before reporting the bug.

4 Evaluation

This section introduces the experiments performed to evaluate the effectiveness and usefulness of MopFuzzer via answering the following research questions.

- **RQ1 (Usefulness):** Can MopFuzzer detect new JVM JIT compiler bugs?
- **RQ2 (Effectiveness):** Can MopFuzzer outperform the state-of-the-arts in terms of code coverage and optimization interactions?
- **RQ3 (Components' Contribution):** How is the contribution of the major components of MopFuzzer?

4.1 Experiment setup

Target JVMs. We selected popular JVM implementations, OpenJDK [38] and OpenJ9 [37], as our test targets. Specifically, we compiled the latest debug builds of the Long-Term Support (LTS) versions (i.e., 8, 11, 17, 21) and the mainline version (i.e., 23) of OpenJDK and OpenJ9 in our experiments.

Baselines. We compare MopFuzzer with two JVM testing technologies *JITFuzz* [50] and *Artemis* [28] since they are the latest state-of-the-art tools. For instance, Artemis reported

Table 2. Status of the reported bugs by MopFuzzer

Category	OpenJDK	OpenJ9	Total
Numbers of reported bugs			
Confirmed	45	14	59
In Progress	19	9	28
Fixed	7	4	11
Duplicate	5	1	6
Not Backportable	14	0	14
Types of reported bugs			
Crash	39	2	41
Miscompilation	6	12	18

Table 3. Distribution of the detected bugs across OpenJDK LTS and mainline versions

Affected Version	JDK-8	JDK-11	JDK-17	JDK-21	Mainline
#Bugs	26	9	13	9	12
#Not Backportable	12	2	0	0	0

its superior performance compared with other baselines recently [28]. Additionally, similar to MopFuzzer, both JITFuzz and Artemis have designed mutators to validate specific optimization techniques. In contrast, other fuzzers employ more generic mutators that do not target specific optimization strategies and hence are excluded as baselines. For fair comparisons, we use the same seed pool (i.e., Regression Test Suites of OpenJDK [42]) for evaluation.

Parameters. Our practice suggests that when exploring the compiler's optimization interactions, the mutants generated after 50 iterations can balance well between cost and effectiveness. Since MopFuzzer introduces loop structures into seed programs, an excess of such structures will significantly impact the execution efficiency of the generated mutants. Therefore, we empirically set the *MAX Iterations* to 50 in our evaluations to test whether the JVM can handle optimization interactions correctly.

Environment. Our evaluation was conducted on one Linux server with two Intel(R) Xeon(R) Gold 6248R CPUs and 256GB RAM. When comparing code coverage between MopFuzzer and baselines, we use the configuration option `--enable-native-coverage`, which enables native compilation with the code coverage data. In particular, we focus on the coverage statistics of HotSpot and ignore other JDK components. When constructing the Optimization Behavior Vector, we set JVM command flags (i.e., -Xcomp -XX:CompileCommand="compileonly,className::methodName") to force compilation of the target method. We also pass the 15 JVM flags (as shown in our website [35]) to record the runtime optimization behaviors.

4.2 RQ1: Usefulness of MopFuzzer

Detected Bugs. We applied MopFuzzer to test the latest debug build of OpenJDK and OpenJ9. During the three months of testing, MopFuzzer has detected 45 bugs in OpenJDK and 14 bugs in OpenJ9. A detailed breakdown of the bug statistics is presented in Table 2. We found one bug (i.e., Issue-18919) can also be detected by Artemis. Other bugs cannot be detected by the baselines since the key code structures triggering the bug are difficult for Artemis and JITFuzz to generate (see Section 4.3). Among all the confirmed bugs, most (41 out of 59) are crashes. Moreover, among the bugs reported in OpenJDK, two are marked with priority P2, 13 with P3, and 30 with P4. Table 5 shows the distribution of detected bugs across OpenJDK LTS and mainline versions. Note that one bug can affect multiple versions. Specifically, MopFuzzer can detect bugs in each LTS version, and it can also identify 12 bugs in the mainline version. Such results demonstrate its effective bug detection capabilities. We notice that some bugs identified in earlier versions (e.g., JDK-8) are commented by developers as non-backportable. For instance, the developer commented on JDK-8324853, "*Fixed in JDKs 19, 17, 11. Not backported to JDK 8.*" Particularly, the developer explained via email "*In that case it's expected that previous JDK releases are affected...In most cases there is probably a good reason for why we decided to not backport.*" We conjecture that the decision might stem from significant code changes between higher and lower versions, making it challenging to backport these bug fixes. Therefore, we classify these bugs as "Not Backportable" and avoid reporting such bugs in subsequent tests. There are 12 bugs in JDK-8 and 2 bugs in JDK-11 that are marked as "Not Backportable".

We also find that all the test seeds triggering bugs involve various JVM optimization behaviors. It is the interaction of these behaviors that challenges the JVM's ability to handle optimizations correctly. For instance, MopFuzzer discovered a P3 level bug JDK-8322743. The bug-triggering case evokes the JVM to perform optimizations related to loops, lock nesting, function inlining, and escape analysis. Eventually, this bug was caused by the incorrect handling of interactions among these optimizations.

Table 4. The distribution of the affected JIT components. "#" denotes the number of bugs detected by MopFuzzer.

HotSpot Component	#	OpenJ9 Component	#
Global Value Number., C2	10	Redundancy Elimination	4
Ideal Loop Optimizat., C2	7	Loop Optimization	3
Code Generation, C2	7	Pattern Recognition	2
Ideal Graph Building, C2	5	Dead Code Elimination	1
Macro Expansion, C2	4	Escape Analysis	1
Cond. Const. Prop., C2	1	SIMD Support	1
Runtime	4	Value propagation	1
Other JIT Compone.	7	Runtime	1

Affected JIT Compiler Components. The 59 confirmed bugs affect different JVM components, which are listed in Table 4. For HotSpot, the majority of bugs impact the C2 compiler. This makes sense since C2 is much more complex

Table 5. The top five mutators and mutator pairs involved in the 59 bug-triggering test cases

Top Mutators	Ratio	Top Mutator Pairs	Ratio
LoopUnroll.	30.5%	LoopUnroll. + LockElim.	13.6%
LockElim.	25.4%	LockElim. + DeReflect.	8.5%
DeReflect.	22.0%	DeReflect. + EscapeAnalys.	8.5%
LoopUnswitch.	16.9%	Deopt. + DeadCodeElim.	8.5%
EscapeAnalys.	16.9%	LoopUnroll. + LockCoarsen.	6.8%

than C1 and employs more aggressive optimizations. The component most affected is *Global Value Numbering*, which is designed to identify expressions in a program that compute the same value. Regarding OpenJ9, there are 4 bugs that affect *Redundancy Elimination*. If this component incorrectly removes certain code, it can easily lead to miscompilation.

Valuable Mutators. To ascertain the effectiveness of various mutators in bug detection, we analyzed the 59 test cases (after reduction) that trigger bugs. Specifically, we find that each mutator is involved in at least one test case, indicating that all proposed mutators are essential. Delving deeper, we find that certain mutators and their combinations are more frequently involved in triggering test cases. Table 5 shows the ratio of the top mutators and mutator pairs involved in the test cases. Specifically, mutators most frequently associated with bugs were *LoopUnrolling-evoke* (30.5%), *LockElimination-evoke* (25.4%), and *DeReflection-evoke* (22.0%). Moreover, the combination of *LoopUnrolling-evoke* and *LockElimination-evoke* was particularly potent, accounting for 13.6% of the cases. Such observations suggest that JVM developers should pay more attention to these optimizations and scenarios to ensure the JVM's correctness.

A Case Study of JDK-8312741. To understand how MopFuzzer maximizes the compiler optimization interactions, we conduct a case study of the mutants that trigger JDK-8312741. In particular, we discover that the 48*th* mutant causes the compiler to crash. Then, we plot the curve to illustrate how the optimization behavior changes during iterations. Specifically, we calculate the Euclidean distance by comparing the OBV of *i*th mutant and the original seed throughout the 48 iterations. The curve is shown in Figure 1. We can find that the curve starts with relatively small values, and escalates to higher values as the iterative process. This pattern suggests that as iterations continue, the mutant will trigger various optimization interactions in the compiler significantly. Second, since the bug is triggered by the 48*th* mutant, this suggests that a certain level of optimization interactions must be accumulated before the bug is triggered. Notably, there are several large jumps (e.g., the 12*th* mutant), implying drastic changes at certain iterations. We speculate that this is caused by the inserted code in these rounds forming more complex code structures with the previously inserted code and the seed's own code, prompting the compiler to handle a greater amount of optimization interactions.

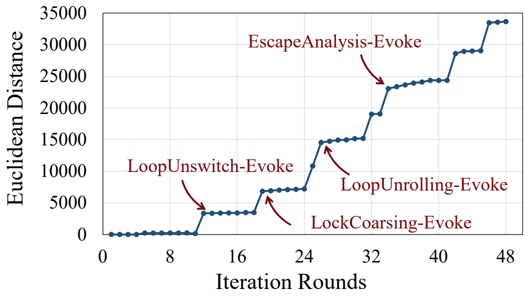


Figure 1. Increment of the Euclidean distance, each point is calculated by comparing the OBV of the *ith* mutant and the original seed. The 48*th* mutant triggers JDK-8312741. We mark some "large jump" points in red.

4.3 RQ2: Effectiveness of MopFuzzer

In this RQ, we compare MopFuzzer with *JITFuzz* and *Artemis* on OpenJDK17 with respect to the achieved *code coverage*, *the increment of optimization behaviors*, and *bug detection capability*. For code coverage, we focus on the JVM's main components: C1, C2, Runtime, and GC (Garbage Collection). We repeat each experiment three times and calculate the average value to minimize the impact of randomness. For the increment of optimization behaviors, we measure the Euclidean distance of OBV between the final mutant and the original seed. The mechanism adopted by such tools are different, and we select the final mutant $c^\star$ for each tool as follows: (1) For MopFuzzer, we consider the 50*th* mutant as $c^\star$; (2) For JITFuzz, the $1,000$*th* mutant (JITFuzz iterates 1,000 rounds for one seed by default) is deemed as $c^\star$; (3) As for Artemis, which does not perform seed scheduling nor mutate seeds iteratively, we use the 1*st* mutant as $c^\star$. Accordingly, each tool processes a different number of seeds within the same time frame. We ensure fairness by setting the same fuzzing time, which is 24 hours.

Code Coverage. Figure 2 shows the results in terms of code coverage, which is averaged over the repeated experiments. We can see that MopFuzzer surpasses both baselines in terms of line coverage. Specifically, it achieves a 63.7% line coverage across the four primary components, while JITFuzz and Artemis attain 62.0% and 62.8% respectively. It is important to highlight that the four main components of OpenJDK17 encompass roughly 126K lines of code. Therefore, an improvement of just 1% in terms of code coverage can result in thousands of new lines being covered. Besides, many studies [20, 24] have shown that higher coverage does not necessarily lead to detecting more bugs. Although MopFuzzer does not stand out in terms of coverage, our focus on maximizing optimization interactions can generate effective seeds that eventually lead to deeper bugs.

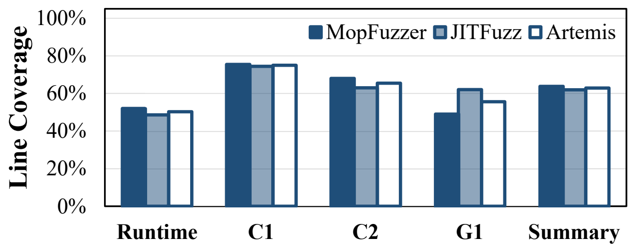


Figure 2. Line coverage achieved by MopFuzzer, JITFuzz, and Artemis. The "Summary" includes the average results for the four components.

Looking into individual components, MopFuzzer outperforms in the C1 and C2 components, achieving the highest coverage of 75.5% and 67.9%. However, JITFuzz excels in the GC component, achieving a high coverage of 62.0%, which is also significantly higher than other tools. In conclusion, the mutants generated by MopFuzzer can prompt the JVM to trigger more optimization behaviors, resulting in higher code coverage, especially with respect to C1 and C2.

The Increment in Optimization Behaviors. Since JITFuzz and Artemis have designed mutators for optimization behaviors, they can also trigger a certain degree of optimization interactions. The boxplot in Figure 3 presents the comparison between MopFuzzer, JITFuzz, and Artemis. We can find that MopFuzzer significantly outperforms JITFuzz and Artemis in enhancing optimization behaviors, with a median Euclidean distance of 3,881. This suggests that MopFuzzer is generally more effective in activating various compiler optimizations. In contrast, while JITFuzz and Artemis also promote optimization behaviors, their impact is less pronounced than that of MopFuzzer. Specifically, although JITFuzz employs certain optimization triggering mutators and also iterates each seed for 1,000 times, it achieves the lowest median Euclidean distance of 1,192. This indicates that the mutants generated by JITFuzz are ineffective in stimulating a wide range of compiler optimizations. In particular, the mutators of JITFuzz involve limited types of optimization behaviors and overlook their interactions, leading to fewer optimization interactions compared to MopFuzzer. Artemis employs three mutation templates targeting method calls, loops, and uncommon traps, which can only trigger optimization interactions between the inserted mutator and the original code structures due to its non-iterative strategy. Besides, its designed mutators do not interact with each other.

Comparison of Bug Detection Capability. To assess the bug detection capabilities of MopFuzzer against the baselines, we evaluate the number of bugs identified by each tool within a 24-hour period using the same seed pool. The results, as shown in Table 6, indicate that MopFuzzer detected a total of 6 bugs compared to 4 by *Artemis* and 2 by *JITFuzz*. Additionally, the bugs detected by MopFuzzer involve a wider range of HotSpot components (i.e., five components). Such results reflect that MopFuzzer outperforms both JITFuzz and Artemis across multiple components of the HotSpot.

Table 6. Comparison of bug detection capability on OpenJDK for all approaches within 24 hours. The number in brackets denote the bugs are uniquely detected by the approach.

Components	MopFuzzer	Artemis	JITFuzz
Global Value Number., C2	2 (2)	0 (0)	0 (0)
Ideal Loop Optimizat., C2	1 (1)	2 (2)	0 (0)
Macro Expansion, C2	1 (1)	0 (0)	0 (0)
Cond. Const. Prop., C2	1 (1)	0 (0)	0 (0)
Regsister Allocation, C2	0 (0)	1 (1)	0 (0)
Ideal Graph Building, C2	0 (0)	1 (1)	0 (0)
Value Mapping, C1	0 (0)	0 (0)	2 (2)
Runtime	1 (1)	0 (0)	0 (0)
Total	**6 (6)**	**4 (4)**	**2 (2)**

MopFuzzer is especially effective in uncovering complex bugs through generating test cases impacting multiple phases of JIT compilations. For instance, detecting JDK-8322743 demonstrates its ability to handle optimizations including escape analysis, lock elimination, autobox elimination, and deoptimization. In contrast, Artemis and JITFuzz focus on fewer optimization strategies. Specifically, Artemis only focuses on designing loop-related mutators aiming to validate the correctness of different JIT compilation choices. It incorporates these mutators to make certain parts of the code "hot", thereby controlling whether these segments are compiled or not by JIT. Even if Artemis were enhanced with the same mutators as provided by MopFuzzer, it cannot sufficiently validate whether the target JVM can handle multiple optimization behaviors correctly. This is because Artemis does not adopt an iterative strategy, and thus these applied mutators would not interact with each other. However, we acknowledge that Artemis-generated test cases feature more complex loop structures, an area that MopFuzzer lacks. In fact, there are multiple ways to design loop-related and other mutators, and in this study, we explore just one way. In the future, we plan to explore additional implementations of mutators to enhance MopFuzzer's bug detection capabilities in specific JIT components.

JITFuzz, focusing merely on escape analysis, function inlining, and simplification, applies mutators iteratively but

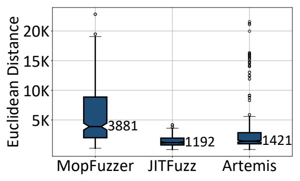


Figure 3. Distribution of Euclidean distance for different approaches

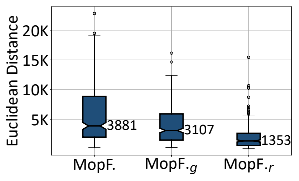


Figure 4. Distribution of Euclidean distance for different variants

not in a nested manner. As a result, complex optimization behaviors involving multiple phases are not sufficiently analyzed, leading to less effective bug detection. Although equipping JITFuzz with MopFuzzer's mutators could potentially enhance its capabilities, its coverage-oriented strategy has been shown to be less effective in uncovering bugs, as indicated in Table 6. Furthermore, the code inserted into the seed programs by JITFuzz is independent of each other, making it challenging to generate complex test cases to check how the JVM handles optimization interactions among various optimization behaviors.

4.4 RQ3: Contribution of Major Designs

In this section, we evaluate the key components of MopFuzzer. Specifically, we design the following two variants and compare them with MopFuzzer in terms of the increment in optimization behaviors and bug detection capability.

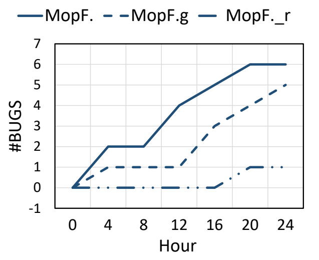
- **MopFuzzer$_g$**: This variant randomly selects a mutator at each iteration without guidance based on profile data. This variant can reveal whether the guidance of profile data can ***accelerate*** the generation of effective mutants, thereby triggering more optimization behaviors.
- **MopFuzzer$_r$**: This variant randomly selects a statement rather than focusing on a fixed mutation point to mutate at each iteration. This variant can reveal whether the mutation generation strategy adopted by MopFuzzer, inserting code ***nested around or adjacent to*** the existing mutated code, can enable the compiler to trigger a greater amount of optimization interactions.

Optimization Behavior Exploration. We employ the same setting as RQ2 to test MopFuzzer and its two variants and measure the Euclidean distance of OBV between the final mutant (i.e., the $50th$ mutant) and the original seed. The boxplots in Figure 4 show the comparison results. We can find that MopFuzzer outperforms its variants in triggering more optimization behaviors in the compiler. In contrast, MopFuzzer$_g$, which disables guidance based on profile data, exhibits a significant decrease in this metric (i.e., 19.9% = (3881-3107)/3881), indicating that the absence of profile-guided mutator selection may lead to less effective mutant generation. We believe the guidance plays a pivotal role in accelerating the generation of effective mutants.

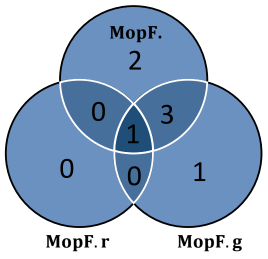
Striking difference is observed for variant MopFuzzer$_r$, which selects a random statement to mutate each time. This variant achieves drastically lower values (i.e., degraded by 65.1% = (3881-1353) / 3881) compared to the other two, suggesting that random statement selection significantly hampers the effectiveness of the tool in triggering optimization interactions. This phenomenon reaffirms our observation: if the code inserted by MopFuzzer, being nested around or adjacent to the existing ones, can effectively enhance the effectiveness of the generated mutants.

Bugs Detection Capability. We further investigate the bug detection capability by comparing MopFuzzer and the two variants within 24 hours. Figure 5a illustrates the number of detected bugs. We can find that MopFuzzer uncovers more bugs as time progresses. In contrast, MopFuzzer$_r$ only detected a limited number of bugs. This shortfall is attributed to the method adopted by MopFuzzer$_r$ to generate new seeds, which do not nest or adjoin optimization evoking mutators, thereby reducing the interactions between different optimization behaviors.

Further, we analyze the overlap in bug detection across different variants, as shown in Figure 5b. In this experiment, two bugs with the same root cause are deemed as the same bug. The data reveals that MopFuzzer can identify nearly all bugs detected by the other variants. There is just one exception: a single bug identified by MopFuzzer$_g$ but missed by MopFuzzer. This finding aligns with our observation that guidance speeds up bug detection in MopFuzzer. Moreover, the fact that MopFuzzer$_g$ can detect about 5/6 of the bugs found by MopFuzzer underscores its strong practicality. Consequently, for the JVM implementations that offer few VM flags, MopFuzzer$_g$ can be applied and can also demonstrate promising performance in detecting bugs.



(a) The number of detected bugs over time by different variants



(b) Overlap of detected bugs of different variants

Figure 5. The bugs detected by MopFuzzer and its variants

5 Discussion

5.1 Limitations of MopFuzzer

MopFuzzer utilizes the profile data to guide the fuzzing process, thus boosting its effectiveness. However, such a strategy is limited by the JVM-provided flags, and thus MopFuzzer records a limited range of optimization behaviors. For instance, the JVM does not offer flags for logging behaviors related to de-reflection, preventing MopFuzzer from capturing such behaviors. This limitation hinders MopFuzzer's ability to flawlessly approximate all optimization interactions. Nevertheless, we find that the bug detection capability of MopFuzzer is not compromised by the limited profile data. In particular, the results in RQ3 show that even with guidance disabled (i.e., MopFuzzer$_g$), it still delivers effective bug detection capability (see Section 4.4), which reflects the practicality of MopFuzzer. Besides, in the white-box testing setting, we can also proactively instrument extra log information in the VMs to record more optimization behaviors.

5.2 The Generalizability of MopFuzzer

Testing Other JIT Compilers. Many modern Language Virtual Machines (LVMs) use JIT compilation to boost execution efficiency. For example, there are V8 Engine [47] for JavaScript, ART [1] for Android, and CPython [9] for Python. We believe the core components of MopFuzzer can be generalized to other LVMs for bug detection. In particular, our design involves various mutators that evoke specific types of JIT optimizations. We categorize these mutators into two groups: language-agnostic and language-specific. Language-specific mutators rely on particular mechanisms of Java, including *LockElimination-evoke*, *Lock Coarsening-evoke*, *Deoptimization-evoke*, *AutoboxElimination-evoke*, and *DeReflection-evoke*. Adapting these mutators is necessary to align with the specific language constructs of the target LVM. For instance, testers for JavaScript Engines could implement *DeReflection-evoke* referring to JavaScript's reflection mechanisms. On the other hand, other mutators that are independent of Java-specific mechanisms fall into the language-agnostic category. These mutators can be applied directly with minimal modifications, as their optimizations are commonly supported by JIT compilers across various LVMs. Furthermore, our strategy of *iteratively applying mutators at a fixed mutation point* has been proven effective (see Section 4.4). Disabling it significantly reduces the number of detected bugs and triggered optimization behaviors.

Testing Non-JIT Compilers. While many modern LVMs utilize JIT compilation to enhance execution efficiency, substantial environments still rely on non-JIT (static or AOT, Ahead-of-Time) compilation. Examples include GCC [16] for C/C++ and R8 Compiler for Android [8]. To test such non-JIT compilers, we can adapt mutators that do not require runtime information, such as *LoopUnrolling-evoke*, *Inlining-evoke*. For example, *LoopUnrolling-evoke* can be used to examine how loop optimizations are handled in static compilation processes. Regarding runtime behaviors, since non-JIT compilers do not compile code at runtime, we can utilize compiler diagnostic information instead. These diagnostics help understand the decisions made during the optimization process. For instance, the option "-fopt-info-loop" in GCC reveals decisions related to loop-related optimizations, and such information can be used to guide the fuzzing process. Therefore, our ideas can be adapted to test non-JIT compilers.

6 Related Work

We introduce works related to JVM or compiler testing.

JVM Testing. As the stability of the JVM gains increasing attention, various techniques for testing the JVM have been developed [6, 7, 23, 26, 28, 50–53]. Classfuzz [7] guides the mutation process of seed files using code coverage information. Classming [6] takes this further by mutating seed files through altering control flows and delving deeper into JVM testing. JavaTailor [53] mutates seed files by inserting ingredients extracted from historical bug-revealing programs. JOpFuzzer [23] tests JIT compiler behavior by combining compiler options with source code mutation. However, none of these approaches focus on optimization interaction. In this study, MopFuzzer originally validates JVM compilers via maximizing optimization interactions. We hope our work can shed light on the JVM testing for future research.

Testing Other Compilers. Plenty of approaches have been proposed to test compilers for various programming languages [5, 54, 55], including C/C++ [4, 12, 29–32, 45, 46], JavaScript [2, 17, 44, 48], Rust [40], etc. Aamodt et al. [29] proposed a method to identify unstable code due to undefined behavior by comparing the compilation results of the same program by different compilers of C/C++. Theodoridis et al. [45, 46] introduced a method to analyze compiler optimization effectiveness and identify missed opportunities using dead code elimination. Wang et al. [48] used an input wrapping template to trigger JIT compilation in JavaScript engines and make tests self-oracle-aware. Bernhard et al. [2] introduced a technique that leverages the relationship between the JavaScript engine's interpreter and its JIT compiler as a mechanism for detecting bugs in JavaScript engines. Sharma et al. [40] developed the random program generator for Rust compiler testing, producing programs that align with Rust's intricate type system and enforce its borrowing and lifetime rules, ensuring well-defined outcomes.

7 Conclusion

This paper presents a novel approach to test JVM JIT compilers, especially focusing on the interactions among various optimization behaviors. Specifically, it employs a set of originally designed optimization-evoking mutators to proactively trigger various optimization behaviors, and then adopts an iterative process to generate mutants guided by the runtime profile data. We substantiate our idea as MopFuzzer, and it has uncovered 59 bugs, of which 45 are in OpenJDK and 14 in OpenJ9. Although MopFuzzer has only implemented a limited number of optimization-evoking mutators and utilized 19 runtime optimization behaviors to guide the generation of mutants, our idea is extensible. In particular, a broader range of optimization evoking mutators can be adapted to test JIT compilers for other language virtual machines.

Acknowledgments

We sincerely thank all anonymous reviewers and our shepherd, Aritra Sengupta, for their valuable feedback and guidance in improving this paper. This work was sponsored by CCF-Huawei Populus Grove Fund (No. CCF-HuaweiSY202308).

A Artifact Appendix

A.1 Abstract

MopFuzzer is a tool for automated testing of the JDK, utilizing various optimization evoking mutators and profile data-based guidance to rigorously test JVM JIT compilers by maximizing optimization interactions. Our artifact includes the source code of MopFuzzer, the mutation seed files required during the execution of MopFuzzer, and a README file describing the installation process and the bugs discovered by the tool.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Optimization evoking mutators and profile data-based guidance.
- **Compilation:** Maven with JDK17
- **Data set:** JDK Regression Tests
- **Run-time environment:** Linux, Windows.
- **Hardware:** Intel x86_64 CPU
- **Metrics:** Bug detection ability, OpenJDK line coverage and Euclidean distance calculated from profile information. More detail metrics settings are listed in the paper.
- **Output:** Java code that can trigger a JDK crash or miscompilation.
- **Experiments:** semi-automated
- **How much disk space required (approximately)?:** 5GB
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes
- **How much time is needed to complete experiments (approximately)?:** 24 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache License, Version 2.0, January 2004
- **Archived (provide DOI)?:** 10.5281/zenodo.11484183

A.3 Description

A.3.1 How to access. Archived content can be downloaded from the DOI Link [36] or accessed via the the Github Link [35].

A.3.2 Hardware dependencies. As some of the experiments need to get the line coverage information of the JDK, a larger hard disk space is required on the hardware for compiling OpenJDK. We recommend that you leave about 20 GB of free disk space for the installation and operation of MopFuzzer and the target JDKs.

A.3.3 Software dependencies. The JDKs under test need to be installed manually, and the rest of the dependencies have been configured in the maven project of the artifact. MopFuzzer needs the debug build of JVM, so users should download the source code of JVM and set the debug flag. Here we take the OpenJDK Mainline as an example.

```
# git clone https://github.com/openjdk/jdk.git
# cd jdk
# bash configure --enable-debug
# make images
```

A.4 Installation

Please obtain the source code for MopFuzzer from the provided the DOI Link [36] or Github Link [35]. MopFuzzer is developed as a Maven project using JDK17. To configure and run MopFuzzer, you can import it directly into your IntelliJ IDEA workspace as a Maven project, or you can use the command *mvn compile* and *mvn package* in the command line to install other dependencies.

A.5 Evaluation and expected results

Detailed instructions on how to run MopFuzzer can be found in the README file, which contains a list of how to install MopFuzzer, how to run it, and a list of bugs found.

Example commands to run MopFuzzerjar file:

```
# differential testing of two target jdks
path/to/java17/bin/java -jar MopFuzzer.jar
  --project_path benchmarks/JavaFuzzer/tests1/
  --target_case Test0001
  --jdk path/to/JDK1/bin/,path/to/JDK2/bin/
  --enable_profile_guide true

# Testing a single jdk using regression seeds
path/to/java17/bin/java -jar MopFuzzer.jar
  --project_path benchmarks/jtreg17/
  --target_case compiler.codegen.TestBooleanVect
  --jdk path/to/targetJDK/bin/
  --enable_profile_guide true
```

It is expected that the generated mutants are stored in the *mutants* directory. If these mutants have exceptions during the run with target JDK, then MopFuzzer logs these exceptions in the log file corresponding to the mutant. In addition, the log file records the mutators used for each mutation and the related commands.

References

[1] Android Runtime (ART) and Dalvik. https://source.android.com/docs/core/runtime, 2024. Accessed: 2024-04.

[2] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. Jit-picking: Differential fuzzing of javascript engines. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 351–364. ACM, 2022.

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

[4] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. Compiler bug isolation via effective witness test program generation. In Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo, editors, *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*, pages 223–234. ACM, 2019.

[5] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *ACM Comput. Surv.*, 53(1):4:1–4:36, 2021.

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

[7] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of JVM implementations. In Chandra Krintz and Emery D. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 85–99. ACM, 2016.

[8] R8 Compiler. https://developer.android.com/build/shrink-code#optimization, 2024. Accessed: 2024-04.

[9] CPython-3.13. https://tonybaloney.github.io/posts/python-gets-a-jit.html, 2024. Accessed: 2024-04.

[10] Xiaoru Dai, Antonia Zhai, Wei-Chung Hsu, and Pen-Chung Yew. A general compiler framework for speculative optimizations using data speculative code motion. In *Proceedings of the 3nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2005), 20-23 March 2005, San Jose, CA, USA*, pages 280–290. IEEE Computer Society, 2005.

[11] Pedro C. Diniz and Martin C. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *Journal of Parallel and Distributed Computing*, 49(2):218–244, 1998.

[12] Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. 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*, pages 1219–1223. IEEE, 2020.

[13] Stephen J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In Richard Johnson, Tom Conte, and Wen-mei W. Hwu, editors, *Proceedings of the 1st IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2003), 23-26 March 2003, San Francisco, CA, USA*, pages 241–252. IEEE Computer Society, 2003.

[14] C1 Flags. https://github.com/openjdk/jdk/blob/master/src/hotspot/share/c1/c1_globals.hpp, 2024. Accessed: 2024-04.

[15] C2 Flags. https://github.com/openjdk/jdk/blob/master/src/hotspot/share/opto/c2_globals.hpp, 2024. Accessed: 2024-04.

[16] GCC. https://gcc.gnu.org/, 2024. Accessed: 2024-04.

[17] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[18] Urs Hölzle, Craig Chambers, and David M. Ungar. Debugging optimized code with dynamic deoptimization. In Stuart I. Feldman and Richard L. Wexelblat, editors, *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), San Francisco, California, USA, June 17-19, 1992*, pages 32–43. ACM, 1992.

[19] Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In Mary Lou Soffa and Evelyn Duesterwald, editors, *Proceedings of the Sixth International Symposium on Code Generation and Optimization (CGO 2008), April 5-9, 2008, Boston, MA, USA*, pages 165–174. ACM, 2008.

[20] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th international conference on software engineering*, pages 435–445, 2014.

[21] JDK-8312744. https://bugs.openjdk.org/browse/JDK-8312744, 2024. Accessed: 2024-04.

[22] JDK-8324174. https://bugs.openjdk.org/browse/JDK-8324174, 2024. Accessed: 2024-04.

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

[24] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *Proceedings of the 22nd international conference on software analysis, evolution, and reengineering (SANER)*, pages 560–564. IEEE, 2015.

[25] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth B. Russell, and David Cox. Design of the java hotspot™ client compiler for java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1):7:1–7:32, 2008.

[26] Stephen C. Kyle, Hugh Leather, Björn Franke, Dave Butcher, and Stuart Monteith. Application of domain-aware binary fuzzing to aid android virtual machine testing. In Ada Gavrilovska, Angela Demke Brown, and Bjarne Steensgaard, editors, *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Istanbul, Turkey, March 14-15, 2015*, pages 121–132. ACM, 2015.

[27] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004.

[28] Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. Validating JIT compilers via compilation space exploration. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 66–79. ACM, 2023.

[29] Shaohua Li and Zhendong Su. Finding unstable code via compiler-driven differential testing. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *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*, pages 238–251. ACM, 2023.

[30] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for C and C++ compilers with yarpgen. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):196:1–196:25, 2020.

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

[32] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: bounded translation validation for LLVM. In Stephen N. Freund and Eran Yahav, editors, *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 65–79. ACM, 2021.

[33] Michael Paleczny, Christopher A. Vick, and Cliff Click. The java hotspot server compiler. In Saul Wold, editor, *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*. USENIX, 2001.

[34] Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Würthinger. An optimization-driven incremental inline substitution algorithm for just-in-time compilers. In Mahmut Taylan Kandemir, Alexandra Jimborean, and Tipp Moseley, editors, *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, pages 164–179. IEEE, 2019.

[35] MopFuzzer Github Repository. https://github.com/CGCL-codes/MopFuzzer, 2024. Accessed: 2024-04.

[36] MopFuzzer Zenodo Repository. https://doi.org/10.5281/zenodo.11484183, 2024. Accessed: 2024-04.

[37] OpenJDK-11 Repository. https://github.com/ibmruntimes/openj9-openjdk-jdk11.git, 2024. Accessed: 2024-04.

[38] OpenJDK-17 Repository. https://github.com/openjdk/jdk17u, 2024. Accessed: 2024-04.

[39] Vivek Sarkar. Optimized unrolling of nested loops. In John Reynders and Alexander V. Veidenbaum, editors, *Proceedings of the 14th international conference on Supercomputing, ICS 2000, Santa Fe, NM, USA, May 8-11, 2000*, pages 153–166. ACM, 2000.

[40] Mayank Sharma, Pingshi Yu, and Alastair F. Donaldson. Rustsmith: Random differential compiler testing for rust. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 1483–1486. ACM, 2023.

[41] K. Shiv, R. Iyer, C. Newburn, J. Dahlstedt, M. Lagergren, and O. Lindholm. Impact of JIT/JVM optimizations on java application performance. In *Proceedings of the 7th Annual Workshop on Interaction between Compilers and Computer Architecture (INTERACT-7 2003), 8 February 2003, Anaheim, CA, USA*, pages 5–13. IEEE Computer Society, 2003.

[42] OpenJDK Regression Test Suites. https://github.com/openjdk/jdk/tree/master/test/hotspot/jtreg, 2024. Accessed: 2024-04.

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

[44] Lili Sun, Chenggang Wu, Zhe Wang, Yan Kang, and Bowen Tang. Kop-fuzzer: A key-operation-based fuzzer for type confusion bugs in javascript engines. In Hong Va Leong, Sahra Sedigh Sarvestani, Yuuichi Teranishi, Alfredo Cuzzocrea, Hiroki Kashiwazaki, Dave Towey, Ji-Jiang Yang, and Hossain Shahriar, editors, *Proceedings of the 46th IEEE Annual Computers, Software, and Applications Conferenc, COMPSAC 2022, Los Alamitos, CA, USA, June 27 - July 1, 2022*, pages 757–766. IEEE, 2022.

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

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

[47] TurboFan. https://v8.dev/docs/turbofan, 2024. Accessed: 2024-04.

[48] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. Fuzzjit: Oracle-enhanced fuzzing for javascript engine JIT compiler. In Joseph A. Calandrino and Carmela Troncoso, editors, *Proceedings of the 32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 1865–1882. USENIX Association, 2023.

[49] HotSpot Office Wiki. https://wiki.openjdk.org/display/HotSpot/PerformanceTacticIndex, 2024. Accessed: 2024-04.

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

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

[52] Jinjing Zhao, Yan Wen, Xiang Li, Ling Pang, Xiaohui Kuang, and Dongxia Wang. A heuristic fuzz test generator for java native interface. In Séverine Sentilles, Barry W. Boehm, Catia Trubiani, and Anne Koziolek, editors, *Proceedings of the 2nd ACM SIGSOFT International Workshop on Software Qualities and Their Dependencies, SQUADE@ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26, 2019*, pages 1–7. ACM, 2019.

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

[54] Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. An empirical study of optimization bugs in GCC and LLVM. *Journal of Systems and Software*, 174:110884, 2021.

[55] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Computing Surveys*, 54(11s):230:1–230:36, 2022.