Supplementary Material

1 Designed Mutators

LoopPeeling-evoke: Loop Peeling is a common optimization technique used in programming to enhance the performance of loops. Specifically, it involves peeling off a part of the loop structure, often the first or last iteration. This approach simplifies certain aspects of the loop or enhances its performance. To proactively trigger this optimization, we design mutator *LoopPeeling-evoke*, which inserts a loop structure that wraps a if statement and the copy of the MP before the original MP. The presence of this if statement prompts the compiler to split the loop into two: one tailored specifically for the condition set by the if statement, and the other to handle the remaining scenarios.

LoopUnswitching-evoke: Loop Unswitching is an optimization technique to enhance the performance of loops. It involves moving a conditional statement outside of a loop if the condition is independent of the loop itself. This optimization can lead to more efficient code by reducing the number of conditional checks within the loop. The *LoopUnswitching-evoke* mutator aims to trigger such loop unswitching optimization in compilers. The instead code contains two nested loops, with the outer loop driven by the variable 'i' and the inner loop by 'j'. Since the switch statement relies on the outer loop's variable 'i' and is unrelated to the inner loop, the compiler can "unswitch" the statement, moving it outside the outer loop. This change means the inner loop is no longer affected by the switch statement, reducing the number of conditional checks per iteration.

Deoptimization-evoke: In the JVM, Deoptimization is a mechanism used to undo previously applied optimizations. During code execution, the JIT compiler transforms bytecode into native machine code to boost efficiency. However, there are times when these earlier optimization assumptions become invalid. When this happens, the JVM needs to reverse (or deoptimize) these optimizations, reverting to a less efficient but more accurate bytecode interpretation. The *Deoptimization-evoke* mutator is designed to trigger such deoptimization in compilers. The inserted code involves a loop and a conditional statement that executes the copy of MP as the loop nears its end. During code execution, the compiler might predict that the condition in the if statement will mostly not be met. So, it might optimize the code to reduce the overhead of checking this condition in most cases. However, when the copy of MP is executed, changes in the code's behavior, like execution time or memory usage, especially if they affect the loop's performance, may require the JIT to reconsider its optimization decisions. As a result, the JVM's optimization based on the previous assumption becomes invalid. It may trigger deoptimization to ensure correct program behavior by reverting to a less optimized but more reliable form of bytecode.

AutoboxElimination-evoke: AutoboxElimination is a JVM optimization technique related to the autoboxing and unboxing of Java primitive types to reduce unnecessary object creation and garbage collection. Autoboxing in Java is the automatic conversion that the Java compiler makes between the primitive types (like int, double) and their corresponding object wrapper classes (like Integer, Double). Unboxing is the reverse process, where the object wrapper class is converted back to a primitive type. The *AutoboxElimination-evoke* mutator ***requires*** a variable or a expression that is primary data type in the MP. Then, MopFuzzer will wrap the variable or the expression with its corresponding wrapper classes (e.g., 'int' to 'Integer').

RedundantStoreElimination-evoke: Redundant Store Elimination optimization focuses on identifying and eliminating redundant memory operations, particularly those that involve storing values in memory. This optimization technique works by detecting instances where consecutive store operations write the same value to the same memory location, or where a value is stored but not used later. By eliminating these redundant stores, the compiler can reduce unnecessary memory write operations, thereby improving the performance of the application. The *RedundantStoreElimination-evoke* mutator ***requires*** the MP is an assignment, and duplicates the MP to create redundant stores.

AlgebraicSimplification-evoke: The Algebraic Simplification optimization technique involves simplifying complex algebraic expressions into simpler forms during JIT compilation to enhance execution efficiency. The *AlgebraicSimplification-evoke* mutator ***requires*** the MP contains arithmetic expressions, we complicate the arithmetic expression in MP by applying inverse algebraic simplification rules. For instance, take the addition expression '$a + b$'. We transform it into '$(a-N)+(N+b)$', where N is a random number. For the multiplication expression '$a*b$', we convert it to '$((a-N)*b+b*N)$'. And for the bitwise operation '$a\&b$', we apply De Morgan's Law to change it into '$(\sim((\sim a)|(\sim b)))$'.

EscapeAnalysis-evoke: Escape Analysis in a JVM context involves the JVM analyzing the code to determine how and where objects escape in a program. For example, the compiler can determine the object is 'No Escape' if the object is used only within the method where it was created. If an object does not escape a method, the JVM can break the object into its individual fields (Scalar Replacement). The JVM then treats these fields as local variables, further optimizing the performance by avoiding the need to create and manage the object as a whole. The *EscapeAnalysis-evoke* mutator ***requires*** the MP contains a primary data type variable or

Table 1. The optimization evoking mutators in the MopFuzzer (8/13). We use the statement $m = a + t.f()$ as the Mutation Point (MP) before mutation. The "Cond" column indicates whether applying this mutator requires a condition. The examples show how the mutators insert or change code on MP. The updated MP (marked with MP_n) will be used for subsequent iterations.

Mutator	Illustration	Cond	Example

LoopPeeling-evoke	Insert a loop structure before the MP. The loop structure wrap a if statement and a copy of MP (only execute a few times).	✗	+ for (int i = 0; i <N; i++) {
			+ if (i < 10) { m = a + t.f(); }}
			  m = a + t.f(); // MP_n

LoopUnswitching-evoke	Insert two nested loops, a switch statement and a copy of MP. The outer loop driven by the variable 'i' and the inner loop by 'j'. Since the switch statement relies on the outer loop's variable 'i' and is unrelated to the inner loop, the compiler can "unswitch" the statement, moving it outside the outer loop.	✗	+ for (int i = 0; i <M; i++) {
			+ for (int j = 0; j <N; j++) {
			+ switch(i) {case -1 : case -2 : case -3 : break;
			+ case 0 : m = a + t.f(); } } }
			  m = a + t.f(); // MP_n

Deoptimization-evoke	Insert a loop structure before the MP. The loop structure wrap a if statement and a copy of MP (only execute when the loop is about to end.)	✗	+ for (int i = 0; i <N; i++) {
			+ if (i == N − 1) { m = a + t.f(); }}
			  m = a + t.f(); // MP_n

AutoboxElimination-evoke	If MP contains a primary data type variable or expression, we wrap the variable or the expression with its corresponding wrapper classes.	✓	− m = a + t.f(); + m = Integer.valueOf(a) + t.f(); // MP_n

RedundantStoreElimination-evoke	If the MP is an assignment, we duplicate the MP to create redundant statements.	✓	+ m = a + t.f();
			  m = a + t.f(); // MP_n

AlgebraicSimplification-evoke	If the MP contains arithmetic expressions, we complicate the expression in MP by applying inverse algebraic simplification rules such as adding and subtracting the same value.	✓	− m = a + t.f(); + m = (a − N) + (N + t.f()); // MP_n

EscapeAnalysis-evoke	If the MP contains a primary data type variable or expression, we will define a custom wrapper class, implemented its constructor and the 'v' method. We then replace the variable or the expression in MP with instances of this custom wrapper class and call the 'v' method to get value.	✓	− m = a + t.f();
			+ m = new MyInteger(a).v() + t.f(); // MP_n
			  ...
			+ class MyInteger{public int v;
			+ public MyInteger(int v){this.v = v; }
			+ int v(){return v; } }

DeadCodeElimination-evoke	Insert a if-statement that wraps the copy of MP, the if-statement condition will evaluate always to false.	✗	+ if (System.currentTimeMillis()<0) {
			+ m = a + t.f(); }
			  m = a + t.f(); MP_n

expression. If so, MopFuzzer will define a custom wrapper class, implemented its constructor and the 'v' method. Then it replace the variable or the expression in MP with instances of this custom wrapper class and call the 'v' method to get value. The instance of the wrapper class does not escape the method so that the Scalar Replacement optimization can be activated.

DeadCodeElimination-evoke: Dead Code Elimination (DCE) is a common optimization technique used in compilers. The DCE involves identifying and removing code that does not affect the program's outcome—code that is never executed or has no impact on the program's results. This process makes programs more efficient by reducing their size and the amount of computation needed. The *DeadCodeElimination-evoke* mutator aims to evoke the DCE optimization proactively in compilers. It inserts a if-statement that wraps the copy of MP, the if-statement condition will evaluate always to false.