MUSEUM: Debugging Real-World Multilingual Programs Using Mutation Analysis

Shin Hong[1], Taehoon Kwak[2], Byeongcheol Lee[3,*], Yiru Jeon[2], Bongseok Ko[3], Yunho Kim[2], Moonzoo Kim[2]

Abstract

Context: The programming language ecosystem has diversified over the last few decades. Non-trivial programs are likely to be written in more than a single language to take advantage of various control/data abstractions and legacy libraries.
Objective: Debugging multilingual bugs is challenging because language interfaces are difficult to use correctly and the scope of fault localization goes beyond language boundaries. To locate the causes of real-world multilingual bugs, this article proposes a mutation-based fault localization technique (MUSEUM).
Method: MUSEUM modifies a buggy program systematically with our new mutation operators as well as conventional mutation operators, observes the dynamic behavioral changes in a test suite, and reports suspicious statements. To reduce the analysis cost, MUSEUM selects a subset of mutated programs and test cases.
Results: Our empirical evaluation shows that MUSEUM is (i) effective: it identifies the buggy statements as the most suspicious statements for both resolved and unresolved non-trivial bugs in real-world multilingual programming projects; and (ii) efficient: it locates the buggy statements in modest amount of time using multiple machines in parallel. Also, by applying selective mutation analysis (i.e., selecting subsets of mutants and test cases to use), MUSEUM achieves significant speedup at the cost of marginal accuracy loss compared to the full mutation analysis.
Conclusion: MUSEUM locates the causes of real-world multilingual bugs accurately and efficiently in a language agnostic manner through mutation analyses. Our light-weight analysis approach would play important roles as programmers write and debug large and complex programs in diverse programming languages.

Keywords: language interoperability, foreign function interface, mutation analysis, debugging

1. Introduction

Modern software systems are written in multiple programming languages to reuse legacy code and leverage the languages best suited to the developers' needs such as performance and productivity. In other words, the feasibility of a single general-purpose language for an entire program becomes low in developing modern complex software for diverse tasks. Over the last few decades, language designers have made a variety of choices in designing the syntax and semantics of their languages. The result is a robust ecosystem where a few languages cover the most use in part due to open source libraries and legacy code while many languages exist for niche uses [1]. This ecosystem encourages developers to write a *multilingual program* which is a non-trivial program written in more than a single language. High-level languages such as Java, Python, and OCaml provide standard libraries, which typically call legacy code written in low-level languages (e.g., C) to interface with the operating system. A number of the projects for the legacy libraries that have evolved for decades provide language bindings for multiple different languages. A large scale software project employs a number of libraries written in multiple languages.

Correct multilingual programs are difficult to write in general due to the complex language interfaces such as Java Native Interface (JNI) and Python/C that require the programs to respect a set of thousands of interface safety rules over hundreds of application interface functions [2, 3]. Moreover, if a bug exists at interactions of code written in different languages, programmers are required to understand the cause-effect chains across language boundaries. Despite the advance of automated testing techniques for complex real-world programs [4, 5, 6, 7, 8], debugging multilingual bugs (e.g., a bug whose cause-effect execution chain crosses language boundaries) in real-world programs is still challenging and consumes significant human effort. For instance, Bug 322222 in the Eclipse bug repository crashes JVMs with a segmentation fault in C as an effect when the program throws an exception in Java as the cause (Section 6.5). Locating and fixing this bug took a heroic debugging effort for more than a year from 2009 to 2010 with hundreds of comments from dozens of programmers before the patch went into Eclipse 3.6.1 in September 2010.

The existing bug detectors targeting multilingual

*Corresponding author.
[1]Handong Global University, hongshin@handong.edu
[2]KAIST, {thkwak, podray, kimyunho}@kaist.ac.kr, moonzoo@cs.kaist.ac.kr
[3]GIST, {byeong, bsk}@gist.ac.kr

bugs [2, 9, 10, 11, 12, 13, 14, 15, 16] are not effective in debugging this case, because they can only report violations of predefined interface safety rules, but cannot indicate the location of the bug, especially when the bug does not involve any known safety rule violations explicitly. Moreover, these bug detectors do not scale well to a large number of languages and various kinds of program bugs since they have to deeply analyze the semantics of each language for each kind of bug.

This article proposes MUSEUM, a mutation-based fault localization (MBFL) technique for locating multilingual bugs in real-world programs. Mutation-based fault localization (MBFL) is an approach recently proposed for locating a code lines that causes a test failure accurately. An MBFL technique takes target source code and a test suite including failing test cases as input, and assesses suspiciousness of each statement in terms of its relevance to the error. To calculate suspiciousness scores, it observes how testing results (i.e., pass/fail) change if the statement is modified/mutated. MUSEUM extends an MBFL technique MUSE [17] which is limited for targeting monolingual bugs (i.e., bugs in C). In contrast to MUSE which mutates only C code in a simple syntactic way, MUSEUM applies new mutation operators that systematically modify the multilingual features/behaviors of a target program (see Section 3.3) and traditional mutation operators together to localize multilingual faults accurately.

Our empirical evaluation on the eight real-world Java/C bugs (Sections 5– 7) demonstrates that MUSEUM locates the bugs in non-trivial real-world multilingual programs far more accurately than the state-of-the-art spectrum based fault localization (SBFL) techniques. MUSEUM identifies the buggy statements as the most suspicious statements for all eight bugs (Section 4). For example, for Bug 322222 in the Eclipse bug repository, MUSEUM indicates the statement at which the developer made a fix as the most suspicious statement among total 3494 candidates (Section 6.5). Furthermore, one case study on an unresolved Eclipse bug (i.e., an open bug whose fix is not yet made) clearly demonstrates that MUSEUM generates effective information for developers to identify and fix the bug (Section 7).

In summary, this paper's contributions are:

1. An automated fault localization technique (i.e., MUSEUM) which is effective to detect multilingual bugs which are known as notoriously difficult to debug.
2. New mutation operators on multilingual behavior which are highly effective to locate multilingual bugs (Section 3.3)
3. Detailed report of the eight case studies to figure out why and how the proposed technique can localize real-world multilingual bugs accurately (Sections 5–7).

This article extends our prior publication [18] in three ways: (i) Section 3.3 elaborates the program mutation with the four additional mutation operators to increase the accuracy of localizing multilingual bugs (ii) Sections 5–6 describe the case studies on the five additional resolved bugs (Bug1,2,3,5,7). Also, Section 7 illustrates a case study on one unresolved open bug (Bug8) to demonstrate how MUSEUM can guide developers to debug a complex multilingual bug (iii) Section 8 shows that MUSEUM can speedup the fault localization process significantly at the cost of marginal accuracy loss by applying selective mutation analysis (i.e., selecting subsets of mutants and test cases to use).

The rest of the paper is organized as follows. Section 2 describes background on multilingual debugging and fault localization techniques. Section 3 presents MUSEUM in detail. Section 4 overviews the empirical study on the eight real-world multilingual bugs. Sections 5 presents the case studies on the bugs that violate the safety rules of language interfaces. Section 6 describes the case studies on general multilingual bugs. Section 7 shows a case study where MUSEUM successfully helped a developer identify a faulty statement to fix an unresolved real-world open bug. Section 8 shows how to reduce the runtime cost of MUSEUM while maintaining high fault localization accuracy by applying selective mutation analysis. Section 9 discusses the observations made through the experiment. Section 10 concludes this paper with future work.

2. Background and Related Work

2.1. Multilingual Bugs

A multilingual program is composed of several pieces of code in different languages that execute each others through language interfaces (e.g., JNI [3] and Python/C). These multilingual programs introduce new classes of programming bugs which obsolete the existing monolingual debugging tools and require much more debugging efforts of programmers than monolingual programs. We classify multilingual bugs into *language interface bugs* and *cross-language bugs.*

2.1.1. Language Interface Bugs

Language interfaces require multilingual programs to follow safety rules across language boundaries. Lee *et al.* [2] classify safety rules in Java/C programs into three classes: (1) state constraints (2) type constraints (3) resource constraints :

- *State constraints* ensure that the runtime system of one language is in a consistent state before transiting to/from a system of another language. For instance, JNI requires that the program is not propagating a Java exception before executing a JNI function from a native method in C.
- *Type constraints* ensure that the programs in different languages exchange valid arguments and return values of expected types at a language boundary. For instance, the `NewStringUTF` function in JNI expects its arguments not to be `NULL` in C.

- *Resource constraints* ensure that the program manages resources correctly. These resource constraints are comparable to the contracts of calling the `free` function for dynamically allocated memory in C. For example, a local reference l to an Java object obtained in a native method m_1 should not be reused in another native method m_2 since l becomes invalid when m_1 terminates [3] (see Section 5.1 as an example of a multilingual bug that violates this resource constraint).

For instance, the manuals for JNI [3] and Python/C describe thousands of safety rules over hundreds of API functions. When a program breaks an interface safety rule, the program crashes or generates undefined behaviors. For instance, a bug of mishandling a Java exception in a native method crashes J9 JVM while HotSpot JVM ignores the propagating exception [2].

2.1.2. Cross-Language Bugs

Cross-language bugs have a cause-effect chain that goes through language interfaces while respecting all interface safety rules. For instance, a program would leak a C object referenced by a Java object that is garbage collected at some point. In this case, the cause of the memory leak is in Java at the last reference to this Java object while the effect is in C (see Section 3.1). On the other hand, the same program would respect all safety rules of language interfaces.

2.2. Debugging Multilingual Bugs

Debugging a program bug consists of the following three steps: (1) detecting an error (2) locating buggy statements (i.e., the code lines responsible for the error) (3) creating a fix on the buggy statements. These three steps are more challenging for multilingual programs than monolingual programs because interfaces and interactions among different languages should be considered, which increases the complexity of debugging.

For the first step (i.e., detecting a multilingual error), there exist dozens of static and dynamic analysis techniques [2, 9, 10, 11, 12, 15, 16]. Some of these techniques provide bug-checkers that detect/predict interface safety rule violations (for example, CheckJNI which is a built-in dynamic JNI checkers in JVMs such as HotSpot and J9).

Unfortunately, few techniques support the second step (i.e., fault localization of multilingual bugs). Although the aforementioned static and dynamic analysis techniques can detect/predict multilingual errors, locating the buggy statements that cause the multilingual errors is still challenging because the root cause of multilingual errors is often non-trivial and located far from the error sites (see Sections 5–6). Although multilingual debuggers may support programmers to locate the causes of the bugs manually [19], it still consumes a considerable amount of time to localize a complex multilingual bug (e.g., Bug 322222 of the Eclipse bug repository).

2.3. Mutation-Based Fault Localization

Fault localization techniques [20, 21] aim to locate the buggy statement that causes an error in the target program (i.e., the second step of debugging) by observing test runs. Fault localization has been extensively studied for monolingual programs both empirically [17, 22, 23]and theoretically [24, 25].

Spectrum-based fault localization (SBFL) techniques infer that a code entity is suspicious for an error if the code entity is likely executed when the error occurs. Note that SBFL techniques are *language agnostic* because they calculate the suspiciousness scores of target code entities by using the testing results (i.e., fail/pass) of test cases and the code coverage of these test cases without complex semantic analyses. However, the accuracy of SBFL techniques are often too low to localize faults in large real-world programs.

To improve the accuracy of fault localization, mutation-based fault localization techniques (MBFL) are proposed recently, which analyze diverse program behaviors by using mutants (i.e., target program versions that are generated by applying simple syntactic code change such as replacing `if(x>10)` with `if(x<10)`). MBFL techniques are also language agnostic since they utilize only the testing results (i.e., fail/pass) of test cases on the original target program and its mutants. Moon *et al.* [17] demonstrate that their MBFL technique (calling it MUSE) is 6.5 times more precise than the state-of-the-art SBFL techniques such as Ochiai and Op2 on the 15 versions of the SIR subjects. The key idea of MUSE is as follows. Consider a faulty program P whose executions with some test cases result in error. Let m_f be a mutant of P that mutates the faulty statement, and m_c be one that mutates a correct statement. MUSE assesses the suspiciousness of a statement based on the following two observations:

- *Observation 1*: *a failing test case on P is more likely to pass on m_f than on m_c.* Mutation is more likely to cause the tests that failed on P to pass on m_f *than* on m_c because a faulty program might be partially fixed by modifying (i.e., mutating) a faulty statement, but not by mutating a correct one. Therefore, the number of the test cases whose results change from fail to pass will be larger for m_f than for m_c.

- *Observation 2*: *a passing test case on P is more likely to fail on m_c than on m_f.* A program is more easily broken by mutating a correct statement than by mutating a faulty statement. Thus, the number of the test cases whose results change from pass to fail will be greater for m_c than m_f.

Note that the aforementioned observations are on multiple statements to compare *relative* suspiciousness of statements among target statements to identify more suspicious statements than the others (e.g., a statement s_1 is more suspicious than s_2 and s_3). Moon *et al.* [17] showed that these observations are valid through the experiments on the 15 versions of SIR subjects (e.g., the number of the failing test cases on P that pass on m_f is 1435.9 times larger than the number on m_c on average). Also note that the observation 2 is important because the observation 2 can serve as a tiebreaker by differentiating statements that are equally suspicious in terms of the observation 1 (see the case study results on Bug 7 (Sect. 5.2.2), Bug 2 (Sect. 6.2.2), Bug 3 (Sect. 6.3.2), and Bug 6 (Sect. 6.5.2)).

There exist a few other MBFL approaches. To localize faults precisely, Zhang *et al.* [26] measure fault-inducing change in regression testing and Papadakis *et al.* [27, 28] measure mutant similarities. In contrast, MUSE utilizes the differences introduced by mutants for fault localization.

3. Mutation-Based Fault Localization for Real-World Multilingual Programs

To alleviate the difficulty of debugging multilingual programs, we have developed a MUtation-baSEd fault localization technique for real-world mUltilingual prograMs (MUSEUM). Section 3.1 shows an motivating example of mutation-based fault localization for a multilingual bug. Section 3.2 describes the fault localization process of MUSEUM. Section 3.3 explains new mutation operators of MUSEUM designed for directly mutating interactions at language interfaces. Section 3.4 overviews the prototype implementation of MUSEUM.

3.1. Motivating Example

3.1.1. Target Program

Figure 1 presents a target Java/C program with a memory leak bug failing the assertion at Line 71 (this example is a simplified version of a real-world bug found in Azureus 3.0.4.2 (Bug1 in Table 2)). The program is composed of source files in C and Java defining three Java classes: `CPtr`, `Client`, and `ClientTest`.

`CPtr` (Lines 2–31) characterizes the peer class idiom [3, p. 123] of wrapping native data structures, which is widely used in language bindings for legacy C libraries. The `peer` field (Line 4) is an opaque pointer from Java to C to point to a dynamically allocated integer object in C. The `CPtr` constructor (Line 9) executes the `nAlloc` native method (Lines 17–21) to allocate an integer object in C and stores the address of the integer object in `peer`. While JVMs automatically reclaim a `CPtr` object once the object becomes unreachable in the Java heap, the clients of `CPtr` are required to dispose manually the integer object by executing `dispose` (Line 12) on the `CPtr` object. If the client does not dispose an `CPtr` object before it becomes unreachable, the peer integer object becomes a unreachable memory leak in C.

`Client` (Lines 34–45) is a client Java class of using `CPtr`. The `m` field (Line 35) holds a reference to a `CPtr` object. `add` (Lines 36–39) and `remove` (Lines 40–45) write/read a value to/from the `CPtr` object respectively. `add` instantiates a `CPtr` object, assigns the reference of the new object to `m`, and then writes a value to the object. `remove` reads the value of the `CPtr` object pointed by `m`, disposes the

```
1 : /* CPtr.java */
2 : public class CPtr {
3 :   static {System.loadLibrary("CPtr");}
4 :   private final long peer;
5 :   private native long nAlloc();
6 :   private native void nFree(long pointer);
7 :   private native int nGet(long pointer);
8 :   private native void nPut(long pointer, int x);
9 :   public CPtr(){peer = nAlloc();}
10:   public int get(){return nGet(peer);}
11:   public void put(int x){nPut(peer, x);}
12:   public void dispose(){nFree(peer);} }
13:
14: /* CPtr.c */
15: #include <jni.h>
16: #include <stdlib.h>
17: jlong Java_CPtr_nAlloc(JNIEnv *env,jobject o){
18:   jint *p;
19:   p =(jint *)malloc(sizeof (jint)); /*Mutant m1*/
20:   return (jlong)p;
21: }
22: void Java_CPtr_nFree(JNIEnv *env,jobject o,jlong p){
23:   free((void *)p);
24: }
25: jint Java_CPtr_nGet(JNIEnv *env,jobject o,jlong p){
26:     return *(jint *)p;
27: }
28: void Java_CPtr_nPut(JNIEnv *env,jobject o,jlong p,
29  jint x){
30:     *((jint *)p) = x;
31: }
32:
33: /* Client.java*/
34: public class Client {
35:   CPtr m = null;
36:   void add(int x){
37:     m = new CPtr(); /*Mutant m2*/
38:     m.put(x);
39:   }
40:   int remove(){
41:     int x = m.get();
42:     m.dispose();
43:     m = null;
44:     return x;  /*Mutant m3*/
45: } }
46:
47: /* ClientTest.java */
48: import java.util.*;
49: public class ClientTest {
50:   static final List pinnedObj=new LinkedList();
51:   public static Object pinObject(Object o){
52:     pinnedObj.add(o);
53:     return o;
54:   }
55:   void passingTest(){ // passing test case
56:     try {
57:       Client d = new Client() ;
58:       d.add(1) ;
59:       assert d.remove() == 1;
60:     } catch(VirtualMachineError e) {
61:       assert false; /*potential memory leak in C*/
62:     }
63:   }
64:   void failingTest(){ // failing test case
65:     try {
66:       Client d = new Client() ;
67:       d.add(1) ;
68:       d.add(2) ;
69:       assert d.remove() == 2;
70:     } catch (VirtualMachineError e) {
71:       assert false; /*potential memory leak in C*/
72:     }
73: } }
```

Figure 1: A Java/C program leaking memory in C after garbage collection in Java

`CPtr` object, deletes the reference to the object, and returns the value of the `CPtr` object.

`ClientTest` (Lines 48–73) is a Java class of driving test cases directly for `Client` and indirectly for `CPtr`. It contains one passing test `passingTest` (Lines 55–63) and one failing test `failingTest` (Lines 64-73). The testing oracle validates a program execution by using (1) the assertion statements (Lines 59 and 69) and (2) the exception handler statements (Lines 61 and 71). The assertion statements at Line 59 and Line 69 validate the program state after executing a sequence of `add` and `remove` by checking if `remove` correctly returns the last value given by `add`. On the other hand, the exception handler statements at Line 60 and Line 70 detect failures at arbitrary locations. For instance, runtime monitors such as QVM [29] and Jinn [2] would throw an asynchronous Java exception either at GC safe points or at language transitions.

3.1.2. Passing Test

`passingTest` executes successfully. It satisfies the assertion statement at Line 59 because both the `CPtr` object and the peer integer object in Java and C are reachable, and `remove` at Line 59 returns `1` stored at Line 58. The runtime monitor does not throw any Java exception indicating a memory leak in C because the native integer object is released in the call to `remove`.

3.1.3. Failing Test

`failingTest` fails at Line 71 because the runtime monitor throws an exception due to a memory leak in C. The test case creates one `Client` object (Line 66) and two `CPtr` objects (Lines 67–68), and two native integer objects. The first native peer integer object is a leak in C heap while all the other objects are reclaimed automatically by garbage collectors and manually by C memory deallocator (i.e., `dispose`). The first `CPtr` object and its peer integer object are created in a call to `add` at Line 67. Both become unreachable after the second call to `add` at Line 68. The `CPtr` object would be garbage collected while the program does not manually execute `dispose` on the unreachable native integer peer object. The runtime monitor would perform a garbage collection and find out the native integer peer object is an unreachable memory leak. This memory leak bug appears because `add` does *not* call `dispose` if `m` already points to a `CPtr` object. Thus, we indicate Line 37 as the buggy statement.

3.1.4. Our Approach

MUSEUM generates mutants each of which is obtained by mutating one statement of the target code. Then, MUSEUM checks the testing results of the mutants to localize buggy statements. For example, suppose that MUSEUM generates the following three mutants m_1, m_2, and m_3 by mutating each of Lines 19, 37, and 44.

m_1, *a mutant obtained by removing Line 19*
This mutation resolves the memory leak as the mutant will not allocate any native memory. However, both test cases fail with the mutant because an access to `p` raises an invalid memory access (at `nGet/nPut` of `CPtr`).

m_2, *a mutant obtained by inserting a statement of pinning the Java reference before Line 37*
This mutation inserts a statement of pinning the object: `ClientTest.pinObject(m);` before Line 37, where `pinObject` stores the Java reference `m` into a global data structure `pinnedObjects` (see `Pin-Java-Object` mutation operator in Table 1).

This mutation intends to prolong the lifetime of the Java object referenced by `m` to the end of the program run. This mutation resolves the memory leak in `failingTest` because the first `CPtr` object will not be reclaimed and, thus, will not leak its peer native integer object. The two test cases pass with the mutant because the mutation does not introduce any new bug.

m_3, *a mutant obtained by replacing the return value with 0 in Line 44*
This mutation replaces the variable `x` with an integer constant 0 at Line 44. This mutation fails the assertion at Lines 59 and 69 since the return value of `remove` is always `0`.

From these testing results, MUSEUM concludes that Line 37 is more suspicious than Line 19 and Line 44 because the failing test case passes only on m_2 and the passing test case fails on m_1 and m_3 (see Step 4 of Section 3.2).

Locating the root cause of this memory leak poses challenges in runtime monitoring and fault localization techniques. Memory leak detectors [30, 31] locate memory leaks and their allocation sites, not the cause of the leaks in general. While some leak chasers [29, 32, 33] locate the cause of memory leak, they do not scale well across language boundaries since they do not track opaque pointers and their staleness values across languages. SBFL techniques cannot localize the bug because both `passingTest` and `failingTest` cover the same branches/statements in their executions. Consequently, SBFL techniques cannot indicate any code element that is more correlated with the failure than the others.

3.2. Fault Localization Process of MUSEUM

Figure 2 describes how MUSEUM localizes faults. MUSEUM takes the target source code and the test cases of the target program as input, and returns the suspiciousness scores of the target code lines as output. MUSEUM has the following basic assumptions on a target program P and a test suite T:

1. Existence of test oracles
   A target program has explicit or implicit test oracle mechanism (i.e., user-specified `assert`, runtime

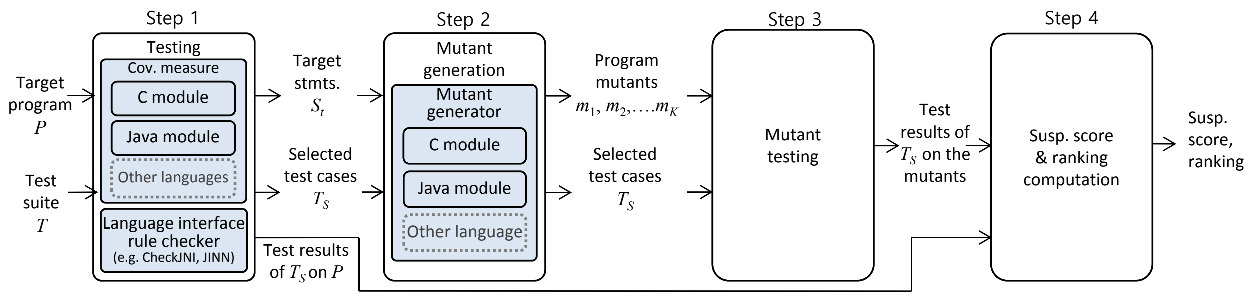
Step 1
Testing
Cov. measure
C module
Java module
Other languages
Language interface rule checker (e.g. CheckJNI, JINN)
Target program P
Test suite T
Target stmts. S_t
Selected test cases T_S
Test results of T_S on P
Step 2
Mutant generation
Mutant generator
C module
Java module
Other language
Program mutants $m_1, m_2, \ldots m_K$
Selected test cases T_S
Step 3
Mutant testing
Test results of T_S on the mutants
Step 4
Susp. score & ranking computation
Susp. score, ranking

Figure 2: Fault localization process of MUSEUM

failure such as null-pointer dereference, and/or runtime monitor such as Jinn [2]) which can detect errors clearly.

2. Existence of a failing test case
A target program has test cases, at least one of which violates a test oracle.

MUSEUM operates in the following four steps:

- **Step 1:** MUSEUM receives P and T and selects target statements S_t and test cases T_S. S_t is the set of the statements of P that are executed by at least one failing test case in T. MUSEUM selects S_t as target statements for bug candidates. Also, MUSEUM selects and utilizes a set of test cases T_S, each of which covers at least one target statement because the other test cases may not be as informative as test cases in T_S for fault localization. To select S_t and T_S, MUSEUM first runs P with T while storing the test results and the test coverage for each test case. Testing results are obtained from the user given assert statements, runtime failures, and multilingual bug checkers such as CheckJNI, Jinn [2], and QVM [29] (Section 2.1).

- **Step 2:** MUSEUM generates mutant versions of P (i.e., $m_1, m_2, ...m_k$) each of which is generated by mutating each of the target statements. MUSEUM may generate multiple mutants from a single statement since one statement may contain multiple mutation points [34]. MUSEUM can localize a bug spanning on multiple statements (not limited for locating a single-line bug). This is because mutating a part of a bug (i.e., one statement among multiple statements that constitute a bug) can still change a failing test case into passing one, which will increase the suspiciousness of the statement constituting the bug [17].

  To reduce the runtime cost, MUSEUM generates only one mutant for every applicable operator at each mutation point. For example, `if(x+2>y+1)` has one mutation point (`>`) for ORRN (mutation operator on relational operator) and two points (`2` and `1`) for CCCR (mutation operator for constant to constant replacement) [34]. MUSEUM generates only one mutant like `if(x+2<y+1)` using ORRN and only `if(x+0>y+1)` and `if(x+2>y+0)` using CCCR.

- **Step 3:** MUSEUM tests all generated mutants with T_S and records the testing results. MUSEUM runs a mutant with a passing test case only if the test case covers the mutated statement. Otherwise, it is obvious that the testing result is the same as the original program. We consider a test fails if the testing time exceeds a given time limit since a mutation may induce an infinite loop. Note that this step can be parallelized on multiple machines for fast fault localization by distributing mutant testing tasks to the multiple machines.

- **Step 4:** MUSEUM compares the test results of T_S on P with the test results of T_S on all mutants. Based on these results, MUSEUM calculates the suspiciousness scores of the target statements of P as follows.

  For a statement s of P, let $f(s)$ be the set of tests that covers s and fails on P, and $p(s)$ the set of tests that covers s and passes on P. Let $mut(s) = \{m_1, \ldots m_k\}$ be the set of all mutants of P that mutates s. For each mutant $m_i \in mut(s)$, let f_{m_i} and p_{m_i} be the set of failing and passing tests on m_i respectively. And let $f2p$ and $p2f$ be the numbers of changed test result from fail to pass and vice versa between P and all mutants of P. The suspiciousness metric of MUSEUM is defined as follows:

$$Susp(s) = \frac{1}{|mut(s)|}\sum_{m_i \in mut(s)}\left(\frac{|f(s) \cap p_{m_i}|}{f2p} - \frac{|p(s) \cap f_{m_i}|}{p2f}\right)$$

  The first term, $\frac{|f(s) \cap p_{m_i}|}{f2p}$, reflects the first observation: it is the proportion of the number of tests that failed on P but now pass on a mutant m_i that mutates s over the total number of all failing tests that pass on a some mutant (the suspiciousness of s increases if mutating s causes failing tests to pass). Similarly, the second term, $\frac{|p(s) \cap f_{m_i}|}{p2f}$, reflects the second observation, being the proportion of the number of tests that passed on P but now fail on a mutant m_i that mutates s over the total number of all passing tests that fail on a some mutant (the suspiciousness of s decreases if mutating s causes passing tests to fail). After dividing the sum of the first term and the second term by $|mut(s)|$, $Susp(s)$ indicates the probability of s to be a faulty statement based on the changes of test results

Table 1: New mutation operators of MUSEUM

No.	Mutation operator	Corresponding language interface rule (Section 2.1)
1	`Clear-pending-exceptions`	State constraints
2	`Propagate-pending-exceptions`	
3	`Throw-new-exceptions`	
4	`Type-cast-to-jboolean`	Type constraints
5	`Type-cast-to-superclass`	
6	`Replace-array-elements-with-constants`	
7	`Replace-target-Java-member`	
8	`Make-global-reference`	Resource constraints
9	`Remove-global-reference`	
10	`Make-weak-global-reference`	
11	`Remove-weak-global-reference`	
12	`Make-local-reference`	
13	`Remove-local-reference`	
14	`Pin-Java-object`	
15	`Switch-array-release-mode`	

on P and $mut(s)$. If a target statement has no mutant (i.e., $|mut(s)|=0$), $Susp(s)$ is defined as 0. MUSEUM defines the first term as 0 if $f2p$ is 0. Similarly, the second term is defined as 0 if $p2f$ is 0. For a concrete example of how to calculate the suspiciousness score of MBFL, see Section II.C of Moon *et al.* [17].

3.3. New Mutation Operators for Multilingual Behavior

In addition to the conventional mutation operators, MUSEUM utilizes new mutation operators to directly mutate interactions at language interfaces and effectively localize multilingual bugs. Specifically, we introduce 15 new mutation operators in Table 1, which change the semantics of a target program regarding the JNI constraints based on the language interface specifications [3, 35] and the previous bug studies [2, 36, 37, 38, 39, 40].

3.3.1. New Mutation Operators for State Constraints

1–3. These mutation operators clear, propagate, or generate a pending exception in a native method to ensure the JVM state constraints. Targets of the three mutation operators are all JNI function calls (i.e., `(*env)->`$<$ *JNIFunction* $>$`(...);`). For example, `Clear-pending-exceptions` clears a pending exception in a current thread by inserting

```
(*env)->ExceptionClear(env);
```

immediately before a JNI function call and immediately after a JNI function call that may throw a Java exception.[4] `Propagate-pending-exceptions` propagates a pending exception to the caller by inserting

```
if((*env)->ExceptionOccurred(env)) return;
```

immediately before a JNI function call and immediately after a JNI function call that may throw a Java exception. `Throw-new-exceptions` creates a new Java exception by inserting

```
Throw_New_Java_Exception(env,
"java/lang/Exception");
```

immediately before a JNI function call and immediately after a JNI function call that may throw a Java exception. The first and the second mutation operators are defined based on a best practice in JNI programming [38] and general solutions for JNI exception bugs [10]. The third mutation operator is motivated by a case of a real-world multilingual bug regarding exception handling across language boundaries [41].

3.3.2. New Mutation Operators for Type Constraints

4. `Type-cast-to-jboolean` explicitly converts an integer expression to `JNI_TRUE` or `JNI_FALSE` when the expression is assigned to a `jboolean` variable. [5] In other words, `Type-cast-to-jboolean` changes an assignment `jbool_var = int_expr;` with

```
jbool_var=int_expr?JNI_TRUE:JNI_FALSE;
```

This mutation operation is motivated by the common pitfall of JNI programming [3, pp.132–133].

5. `Type-cast-to-superclass` changes a JNI call that gets the reference of a class of a given object to get the reference of the superclass of the class by mutating `jclass cls = (*env)->GetObjectClass(env,obj);` with

```
jclass cls=(*env)->GetSuperclass(env,
((*env)->GetObjectClass(env,obj)));
```

This mutation operator is motivated by a report of a real-world bug found in Eclipse 3.4 [2].

6. `Replace-array-elements-with-constants` replaces a Java array reference with another constant Java array. This mutation operator changes a Java array reference used at a JNI function call to the reference to the predefined constant array. For example, this mutation operator change `(*env)->GetIntArrayElements(env, arr, null);` into

```
(*env)->GetIntArrayElements(env,
IntConstArr, null);
```

This mutation is inspired by a real-world bug with an incorrect array data transfer from Java to C [42].

[4] 154 among total 229 JNI functions may throw an exception [3].

[5] `jboolean` is an 8 bit integer type. If a 32 bit integer value is assigned to a `jboolean` variable, the variable can have an unintended Boolean value due to the truncation (e.g., `jboolean_var = 256` will make `jboolean_var` as false).

7. `Replace-target-Java-member` replaces a target field in a class member access with the field of a different class member with the same type, by mutating `(*env)->GetFieldID(env, class, NAME1, SIG);` with

   ```
   (*env)->GetFieldID(env, class,
   NAME2, SIG);
   ```

   where `NAME1`, `NAME2`, and `SIG` are the strings of the original and the changed field names and their type signature, respectively. This mutation operator is motivated by a common pitfall in JNI programming [3, pp.131–132].

3.3.3. New Mutation Operators for Resource Constraints

8–13. These mutation operators increase or decrease the life time of a reference to a Java object (and probably the life time of the referenced Java object too). For example, `Make-global-reference` increases the life time of a local reference l by making the reference as a global one. In other words, `Make-global-reference` inserts the following statement after an assignment statement to a local reference l (i.e., l = *expr*):

   l `= (*env)->NewGlobalRef(env,`l`);`

   In contrast, `Remove-global-reference` decreases the life time of a global reference g (and probably the referenced Java object too) by inserting the following statement for a global reference g:

   `(*env)->DeleteGlobalRef(env,`g`);`

   We have developed four other mutation operators for local references and weak global references. These mutation operators are related to a bug fix pattern regarding reference errors in native code [39].

14. `Pin-Java-object` prevents garbage collectors from reclaiming a Java object by placing a Java reference to the object into a class variable in Java before a reference to the object is removed by an assignment statement. Before an assignment statement `x = obj;`, the mutation operator inserts a statement:

   ```
   Test.pinnedObjects.add(x) ;
   ```

   where `Test.pinnedObjects` is a Java class variable of a list container type. The Java object pointed by `x` is transitively reachable from the class variable, and Java garbage collectors cannot reclaim the object. This mutation operator intends to extend the lifetime of Java objects in a target program and influence interactions of Java and native memory management. This mutation operator is inspired by a safe memory management scheme of SafeJNI [40].

15. `Switch-array-release-mode` alternates the release mode of a Java array access. The release mode decides whether an updated native array will be copied back to the Java array or discarded. For every `(*env)->Release<Type>ArrayElements(env, arr, elems, mode)`, this mutation operator changes the `mode` value from `0` to `JNI_ABORT`, or vice versa. This mutation operator is motivated by a best practice in JNI programming [38].

3.4. Implementation

We have implemented MUSEUM targeting programs written in Java and C (support for other languages will be added later). MUSEUM is composed of the existing mutation testing tools for C and Java, together with the fault localization module that analyzes testing results and computes suspiciousness scores. MUSEUM consists of 1,500 lines of C/C++ code and 1,802 lines of Java code. MUSEUM uses gcov and PIT [43] to obtain the coverage information on C code and Java code of a target program, respectively.

MUSEUM uses the existing mutation tools Proteum/IM 2.0 [44] for C and PIT version 0.33 for Java bytecode together with the 15 new mutation operators for multilingual behaviors (Section 3.3). Proteum/IM implements 107 mutation operators defined in Agrawal *et al.* [34]. Among the 107 mutation operators, MUSEUM uses 75 mutation operators that change only one statement. PIT implements 14 mutation operators all of which are used by MUSEUM. Among the 15 new mutation operators, 14 new mutation operators for C code are implemented with Clang version 3.4 [45], and the one new mutation operator for Java (i.e., `Pin-Java-object`) is built with the ASM bytecode engineering tool version 3.3.1 [46].

4. Experiment Setup and Result

We have evaluated the effectiveness of MUSEUM on the eight bugs in four real-world multilingual software projects. Section 4.1 describes the experiment setup and Section 4.2 presents the fault localization results. The full experiment data and the target program code are available at `http://swtv.kaist.ac.kr/data/museum.zip`.

4.1. Experiment Setup

4.1.1. Real-world Multilingual Program Bugs

Table 2 presents the eight multilingual bugs in four real-world software projects with their programs, symptoms, line of code (LOC) in Java and C, the number of the test cases used to localize the fault, and bug reports or bug-fixing revisions of the target programs. Azureus is a popular P2P file-sharing application. Sqlite-jdbc is a Java Database Connectivity (JDBC) library to access the SQLite relational database management system written in C. Java-gnome is a set of language bindings for the Java

Table 2: Target multilingual Java/C bugs, their symptoms, sizes of the target code, the number of test cases used, and references

Bug	Target program	Symptom	Size of target program				# of TC used	Bug report or bug-fixing revision
			Java		NativeC			
			Files	LOC	Files	LOC		
Bug1	`Azureus` 3.0.4.2	Memory leak in C	2,705	340.6K	N/A	N/A	8	Rev. 1.64 of ListView.java [47]
Bug2	`sqlite-jdbc` 3.7.8	Assertion violation in Java	20	4.6K	3	1.8K	150	Issue 16 [48]
Bug3	`sqlite-jdbc` 3.7.15	Assertion violation in Java	19	4.2K	2	1.7K	159	Issue 36 [49]
Bug4	`java-gnome` 4.0.10	Invalid JNI reference in C	1,097	64.2K	496	65.6K	170	Bug 576111
Bug5	`java-gnome` r-658	Segmentation fault in C	1,134	67.1K	514	69.2K	184	Subversion revision 659 [50]
Bug6	`SWT` 3.7.0.3	Segmentation fault in C	582	118.7K	29	43.3K	50	Bug 322222
Bug7	`sqlite-jdbc` 3.6.0	Exception state violation in C	25	4.9K	2	0.6K	112	`UDFTest` bug in Blink [51]
Bug8	`SWT` 4.3.0	Segmentation fault in C	591	126.6K	29	48.5K	204	Bug 419729 [52]

programming language for use in the GNOME desktop environment. SWT (Standard Widget Toolkit) is an Eclipse widget toolkit for Java to provide user-interface facilities. We selected these projects as target projects because these projects have multilingual bugs that had been analyzed by other practitioners and researchers.

As described in the assumption 1 for fault localization (Section 3.2), the bug reports and commit logs in the last column describe the symptoms of the target bugs so that our test oracle detects test failures. A corresponding bug report indicates both buggy version and its fixed version. All target programs are written in Java and C except `Azureus`. While `Azureus` is a pure Java program, it triggers a memory leak in C when it misuses the application program interface of the Eclipse SWT library written in Java and C.

4.1.2. Test Cases

Regarding test cases, we have used the test cases maintained by the developers of the target programs. We utilize the test cases of the fixed version, at least one of which reveals the target bug in the buggy version (see the assumption 2 in Section 3.2). If the fixed version does not have a test case that fails on the buggy version, we create a failing test case based on the bug report. In addition, to localize a fault accurately, we focus to localize one bug at a time by building a new test suite out of the original test suite. The new test suite consists of one failing test case and all passing test cases that cover at least one statement executed by the failing test case.

4.1.3. System Platform

The experiments were performed on the 30 machines equipped with Intel i5 3.4 GHz with 8 GB main memory (we performed experiment on one core per machine). All machines run Ubuntu 8.10 32-bits, gcc 4.3.2, and OpenJDK 1.6.0. MUSEUM distributes tasks of testing each mutant to the 30 machines. We set the time limit (10 seconds) for each test run on a mutant to avoid the infinite loop problem caused by mutation. Time taken to execute a test run was less than one second on the eight subjects on average.

4.2. Experiment Results

Table 3 reports the experiment data on the eight bugs. The second row shows the number of the source target lines executed by the failing test case (see Step 1 of Section 3.2). The third row shows the total number of the mutants generated by MUSEUM, and the fourth row describes the total number of the target lines on which at least one mutant is generated. The fifth and sixth rows show the number of the mutants on which testing results have changed. The last row describes the runtime cost.

For example, to localize Bug4, we built a test suite containing one failing test case and 169 passing test cases out of the original test suite (see the eighth column of the fifth row of Table 2). MUSEUM generated 718 mutants (at least one mutant for 71% of the target lines (=132/186)). Among the 718 mutants, there are two mutants on which the failing test case passes (see the sixth row of Table 3). [6] We call such mutants as "partial fix" because the failing test case passes on the mutant (but passing test cases may fail on these mutants). The table shows that only 0.28% of the mutants are partial fixes (=2/718). Note that partial fix mutants at s can largely increase the suspiciousness score of s since partial fix mutants increase the numerator of the first term of the suspiciousness formula whose denominator $f2p$ is usually small (e.g., 2 for Bug4) (see the formula in the Step 4 of Section 3.2). Regarding the time cost, MUSEUM takes 25 minutes to localize Bug4 using 30 machines.

Table 4 compares the fault localization results of MUSEUM and the cutting-edge SBFL techniques including Jaccard [53], Ochiai [54], and Op2 [55]. Each entry reports the suspiciousness ranking which is the maximum number of the statements to examine until finding the faulty statement described in the bug report. The percentage number in the parentheses indicates the normalized ranking of the faulty statement out of the total target statements (i.e., $\frac{\text{ranking}}{\text{\# of the target statements}}$). The second row of the table clearly shows that MUSEUM accurately identifies the buggy statement. MUSEUM ranks the buggy statements in Bug1, Bug3, Bug4, Bug7, and Bug8 as the

[6] The number of mutants that make the failing test case pass is equal to $f2p$ since the test suite contains only one failing test case in our experiments.

Table 3: Overview of the experiment data

	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8
# of the target lines	1,939	299	443	186	186	3,494	294	4,998
# of mutants	2,861	691	965	718	369	9,479	844	14,490
# of lines which have a mutant	1,575	219	327	132	103	2,524	226	3,855
# of mutants that make a passing test case fails	305	462	681	364	311	3,044	542	8,766
# of mutants that make a a failing test case passes	1	3	7	2	51	32	3	1
Time cost (in minutes)	12	60	45	25	23	175	50	511

most suspicious statements (i.e., the first ranking). Even for Bug2, Bug5, and Bug6, MUSEUM identifies the buggy statement as the most suspicious statement with the other one, seven, and two statements together (e.g., for Bug5, the suspiciousness scores of the eight statements including the buggy statement are equal). Thus, from these experiments, we conclude that MUSEUM localizes a multilingual bug accurately.

In contrast, SBFL techniques fail to localize multilingual bugs accurately. For Bug6, Op2 ranks the buggy statement as the 3,494nd among the 3,494 target statements (see the fifth row of Table 3), which means that a developer has to examine all target statements to identify the faulty statement.

4.3. Threats to Validity

A major external threat to validity is that the experiment uses a limited number of target programs. To limit this threat, we chose the target subjects that include both language interface bugs and cross-language bugs, and have different symptoms and various related language features. Also, we collected these target programs from various real-world projects used by the related work.

Another threat is that the test cases used in the experiments are limited. To limit this threat, we utilized all available test cases in the real-world target subjects (except Azureus that has no test cases for Bug 1).

A construct threat is that there may be statements that can be recognized as buggy statements other than the ones indicated by the bug reports/fixes used in the studies. Although there might be other buggy statements, we believe that the conclusions still hold because MUSEUM localized the buggy statements reported by the bug reports/fixes as most suspicious ones.

Possible internal threats are that the target programs may have unidentified nondeterminism and/or the MUSEUM tool may have faults. To limit these threats, we carefully reviewed the target programs, the MUSEUM tools, and the experiment results. For further analysis, we have released the full experiment data and the target program code at http://swtv.kaist.ac.kr/data/museum.zip.

5. Case Studies with Language Interface Bugs

Language interface bugs violate one of the three classes of safety rules on language interface [2]: state constraints, type constraints, and resource constraints. We perform two case studies to illustrate how MUSEUM locates the causes of the bugs of violating resource constraints in Section 5.1 and state constraints in Section 5.2. We do not include a case study on a type constraint bug since flow-insensitive multilingual type inference systems [37] can be more suitable to validate type constraints.

5.1. Bug4: Invalid JNI Reference in Java-gnome

This case study illustrates how MUSEUM localizes the cause of dangling JNI references (Bug4) accurately by using the new mutation operators (Table 1).

5.1.1. Bug Overview

Dynamic error detectors [2] detect Bug4 and report the calling context at the failure using the dangling JNI reference as an argument to a JNI function. However, they cannot report the cause location where the JNI reference was stored into a callback object in C heap, which occurs at Line 524 of `binding_java_signal.c` as indicated as the buggy statement in the bug report:

```
387: GClosure* bindings(JNIEnv *env,
     jobject handler,  jclass receiver, ... ) {
...
524:    bjc->rec = receiver;
... }
```

When `bindings` at Line 387 is invoked, the `receiver` parameter is assigned with a local JNI reference. Line 524 stores the local reference in a data structure in the C heap pointed by `bjc`. However, once `bindings` returns back to

Table 4: The ranking of the buggy line identified by MUSEUM and the SBFL techniques

	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8
MUSEUM	1 (0.1%)	2 (0.7%)	1 (0.2%)	1 (0.1%)	8 (4.3%)	3 (0.2%)	1 (0.2%)	1 (0.02%)
Jaccard	80 (4.1%)	4 (1.3%)	5 (1.1%)	83 (44.6%)	61 (32.8%)	3,494 (100.0%)	84 (17.5%)	574 (10.2%)
Ochiai	80 (4.1%)	4 (1.3%)	5 (1.1%)	83 (44.6%)	61 (32.8%)	3,494 (100.0%)	84 (17.5%)	574 (10.2%)
Op2	80 (4.1%)	4 (1.3%)	5 (1.1%)	83 (44.6%)	61 (32.8%)	3,494 (100.0%)	84 (17.5%)	574 (10.2%)

Java, the local reference stored in `bjc->rec` is not valid anymore (i.e., becoming a dangling reference, see the resource constraints in Section 2.1). Later, when the program calls a JNI function with an argument containing the dangling reference, the program crashes with a JNI invalid argument error.

5.1.2. Detailed Experiment Result

MUSEUM localizes the faulty statement accurately by ranking Line 524 as the most suspicious statement (i.e., the first rank without a tie). Table 5 describes the nine mutants ($m1$ to $m9$) that are generated by mutating Line 524. The second column shows the changed statement of each mutant. The third and the forth columns report the number of tests that covered Line 524 and failed on the original program but pass on the mutant (i.e., $|f(s) \cap p_m|$), and the number of tests that covered Line 524 and passed on the original program but fail on the mutant (i.e., $|p(s) \cap f_m|$), respectively (Section 3.2). $m1$, $m2$, $m4$, $m6$, $m8$, and $m9$ are generated by applying our new multilingual mutation operators in Table 1. These mutants are generated by inserting the statements of changing the life time of JNI references right after the target statement. $m3$, $m5$ and $m7$ obtained by applying Proteum terminate the control flow at the level of procedure, statement, and whole program.

In the testing runs, our new mutation operators prevent mutated programs $m1$ and $m4$ from crashing with the failing test case (i.e., `Make-weak-global-reference` and `Make-global-reference` in Table 1, respectively). $m1$ and $m4$ change the failing test case into a passing one (the third column) because they keep `bjc->rec` to store a weak global reference and a global reference respectively and eliminate the dead reference problem caused by the short-lived local reference. On the other hand, the conventional mutation operators (i.e., $m3$, $m5$, and $m7$) do not affect the test results. $m1$ and $m4$ make the first term of the MUSEUM suspiciousness metric large and increase the suspiciousness score of Line 524 significantly because the denominator of the first term is small (i.e.,$f2p$=2) (Section 3.2). In contrast, each of the mutants $m5$ to $m9$ make two passing test cases fail (the fourth column), which increases the second term but in only limited degree due to the large denominator (i.e., $p2f$=6053).

Among the 186 target statements, only Line 524 has partial fix mutants with regard to the given test cases and the given test oracle. Consequently, Line 524 has the highest suspiciousness score due to the new mutation operators which generate partial fixes. Thus, through the case study on Bug4, we confirm that the new mutation operators such as `Make-global-reference` can increase the accuracy of MUSEUM.

In contrast, the SBFL techniques rank the buggy statement as the 83rd suspicious one among the 186 target statements. Such poor result is due to the two coincidentally correct test cases (CCTs) that execute Line 524 but pass because the target program does not use `bjc->rec` as an argument to a JNI function call later with these test cases. Thus, the SBFL techniques consider that Line 524 has low correlation with the failure and assign low suspiciousness score to Line 524.

Note that these CCTs do not make adverse effect to MUSEUM. This is because the mutants (i.e., $m1$ to $m9$) obtained by mutating the buggy statement (i.e., Line 524) do not make these CCTs fail because the mutants and the target program do not use `bjc->rec` as an argument to a JNI function call later with these CCTs. Thus, these CCTs do not increase the second term of the MUSEUM suspiciousness metric (Section 3.2) and do not lower the suspiciousness score of the buggy statement.

5.2. Bug7: JNI Exception State Violation in Sqlite-jdbc

This case study illustrates how MUSEUM localizes the cause of violating exception state constraints (Bug7) accurately by using the new mutation operators (Table 1).

5.2.1. Bug Overview

Bug7 violates a safety rule on language interface that the native code must not invoke a JNI function while the current thread is propagating a pending Java exception. Specifically, consider Lines 183 and 184 of `NativeDB.c` in the `sqlite-jdbc 3.6.0` source release:

```
/* sqlite/src/main/java/org/sqlite/NativeDB.c */
154: static xCall(...) {
..
183:  (*env)->CallVoidMethod(env, func, method) ;
184:  (*env)->SetLongField(env, func, ...) ;
```

Table 5: The nine mutants generated by mutating the buggy statement of Bug4

No.	Mutant generated by mutating Line 524 of `bindings_java_signal.c`	$\lvert f(s) \cap p_m \rvert$	$\lvert p(s) \cap f_m \rvert$
$m1$	`bjc->rec=receiver;` `bjc->rec=(*env)->NewWeakGlobalRef(env,bjc->rec);`	1	0
$m2$	`bjc->rec=receiver;` `bjc->rec=(*env)->NewLocalRef(env,bjc->rec);`	0	0
$m3$	`return; // return back to the caller.`	0	0
$m4$	`bjc->rec=receiver;` `bjc->rec=(*env)->NewGlobalRef(env,bjc->rec);`	1	0
$m5$	`; // remove a statement at Line 524`	0	2
$m6$	`bjc->rec=receiver;` `(*env)->DeleteGlobalRef(env, bjc->rec);`	0	2
$m7$	`kill(getpid(), 9); //terminate the process`	0	2
$m8$	`bjc->rec=receiver;` `(*env)->DeleteLocalRef(env, bjc->rec);`	0	2
$m9$	`bjc->rec=receiver;` `(*env)->DeleteWeakGlobalRef(env,bjc->rec);`	0	2

In an erroneous run, the native code at Line 183 invokes a Java method identified by the `method` argument, which throws a Java exception and abruptly returns to the native code. Then, the current thread is propagating a pending Java exception, and the call statement at Line 184 executes the `SetLongField` JNI function. These event series of throwing Java exception and calling a JNI function violate the exception state rule. The semantics of the `SetLongField` JNI function is left undefined, and JVMs exhibit a variety of undesirable behaviors such as crash [51].

The bug fix checks and clears explicitly the pending Java exception before calling the `SetLongField` JNI function with the following updates:

```
/* sqlite/src/main/java/org/sqlite/NativeDB.c */
154: static xCall(...) {
..
183:  (*env)->CallVoidMethod(env, func, method) ;
+++   if((*env)->ExceptionCheck(env))
+++     xFunc_error(context,env);
184:  (*env)->SetLongField(env, func, ...) ;
```

The conditional part of the inserted statement examines whether or not a Java exception is pending. When a Java exception is pending, the true branch statement activates the auxiliary routine `xFunc_error`, which calls the `ExceptionClear` JNI function to clear the pending Java exception and records this error state. Then, the native code executes the `SetLongField` JNI function without violating the JNI exception state rule.

5.2.2. Detailed Experiment Result

We use the 112 tests cases in the Xerial SQLite JDBC regression test suite. Our test oracle passes 111 test cases and fails one test case due to Bug7. MUSEUM successfully finds the location at which the developer inserts the new code to fix the bug as the most suspicious statement. Table 6 shows the mutants generated from the top four most suspicious statements in order of their suspiciousness ranking. Line 184 of `NativeDB.c` has the highest suspiciousness score because it has two fail-to-pass test runs. Similarly, Line 183 of `NativeDB.c` is ranked as the second most suspicious statement because it has one fail-to-pass test run. No other statements have a fail-to-pass test run (see the sixth row of Table 3). Note that the three partial fix mutants on Lines 183–184 do not fail with the failing test case because the executions of the failing test case do not call an JNI function with a pending exception. The statements of the fourth and the fifth rows of Table 6 are ranked as the 114th together with other 110 statements on which their test results do not change at all.

6. Case Studies with Cross-Language Bugs

Cross-language bugs have their cause-effect chains across a language boundary while respecting all safety rules on language interface. To demonstrate how MUSEUM locates the causes of these cross-language bugs, this section presents case studies for the following five bugs: Bug1 in Section 6.1, Bug2 in Section 6.2, Bug3 in Section 6.3, Bug5 in Section 6.4, and Bug6 in Section 6.5.

Table 6: Four most suspicious statements of the Xerial SQLite JDBC target code (Bug7)

Rank	Susp. score	Statement	Mutant	$\|f(s) \cap p_m\|$	$\|p(s) \cap f_m\|$
1	0.111	`/* NativeDB.c:184 */` `(*env)->SetLongField(env,...);`	`if ((*env)->ExceptionOccured(env)) return; (*env)->SetLongField(env,...);`	1	16
			`return ;`	1	16
2	0.055	`/* NativeDB.c:183 */` `(*env)->CallVoidMethod(env,...);`	`return ;`	1	64
114	0.0	`/* Conn.java:81 */` `this.url = url;`	`Test.pinnedObjects.add(url);` `this.url = url;`	0	0
114	0.0	`/* Conn.java:188 */` `checkCursor(rst, rsc, rsh);`	`; // remove a statement at Line 188`	0	0

6.1. Bug1: Memory Leak Bug in Azureus

Bug1 has its effect of leaking memory in C while the cause is due to missing calls to the routines of releasing resources in Java.

6.1.1. Bug Overview

Azureus 3.0.4.2 has a memory leak bug that Azureus may allocate native memory (i.e., heap objects in C) for an `Image` object in Java, but never de-allocate the native memory. In more detail, `Image` class uses native C methods to allocate/de-allocate native memory for its member objects. Such allocated native memory should be explicitly freed by calling `dispose` of an `Image` object before the object is garbage collected.

To fix this memory leak bug, CVS Revision 1.64 of `ListView.java` inserts a call to the `Image.dispose` method into the `Image.handleResize` method to release the native objects referenced from the `imgView` instance field before Line 523 from which the old `Image` object referenced by `imgView` is not referenced/used anymore:

```
496:public void handleResize(boolean bForce) {
...
508:  if (imgView == null || bForce){
...
523:    imgView = new Image(...) ;
```

Locating the cause of this bug is non-trivial because the cause and effect of this bug appear in different languages, and the incoming references to the leaking native memory are implicitly eliminated after the owner `Image` objects are garbage collected [29].

6.1.2. Detailed Experiment Result

Since Azureus code has no test case, we created one failing test case to exercise the memory leak bug and seven passing test cases to cover reasonable fraction of the source files. To detect the memory leak failure, our test oracle calls the `Image.isDisposed` method and validates the native objects are released after the `Image` objects are garbage-collected. Specifically, our test oracle instruments the `Object.finalize` method to place our validation checks:

```
public final class Image ... {
  protected void finalize() throws Throwable {
    if(!isDisposed()) reportLeak();
    super.finalize();}}
```

To ensure that JVMs perform garbage collections and report memory leaks before terminating each test run, our test cases invokes the `System.gc` method of triggering garbage collections manually.

MUSEUM identifies Line 523 of `ListView.java` as the most suspicious statement (i.e., the first rank). The second column of Table 7 shows that Line 523 has four mutants: $m1$, $m2$ and $m3$, and $m4$. $m1$, $m2$ and $m3$ are obtained from the conventional mutation operators of PIT which replace the target statement with the statements in the table. $m4$ is obtained from our new mutation operator `Add-Java-reference` designed for multilingual bugs (particularly targeting resource constraints, see Table 1) to extend the life time of Java objects.

$m4$ turns the failing test case into a passing one (the third column) because it adds an extra reference to the old object pointed by `imgView`, which prevents the old object from being garbage collected. As a result, the native memory leak does not occur. $m4$ makes the first term of the MUSEUM suspiciousness metric large and significantly increases the suspiciousness score of Line 523. In contrast, $m1$ and $m3$ make two and four passing test cases as failing ones (the fourth column), which increases the second term but in only limited degree due to the large denominator (i.e., $p2f$=1961).

Among the 1939 target statements, only Line 523 has a partial fix mutant (i.e., $m4$) with regard to the given test cases and the given test oracle. Consequently, the line 523 has the highest suspiciousness score. Thus, through the case study on Bug1, we confirm that the new mutation operators such as `Add-Java-reference` can increase the accuracy of MUSEUM.

Table 7: The four mutants generated at the buggy statement of Bug1

No.	Mutant generated from "`imgView = new Image( listCanvas.getDisplay(), clientArea);`" at Line 523 of `ListView.java`	$\|f(s) \cap p_m\|$	$\|p(s) \cap f_m\|$
$m1$	`imgView = null;`	0	2
$m2$	`imgView = new Image(null, clientArea);`	0	0
$m3$	`new Image(listCanvas.getDispaly(),clientArea);`	0	4
$m4$	`global_ref_list.add(imgView);`	1	0

6.2. Bug2: Incorrect Call to Java Native Methods in Sqlite-jdbc

Bug2 has its cause-effect chain across Java and C and fails the Java assertion statements. Bug2 is a nontrivial semantic bug to track down its cause since testers must follow the flow of data values along multilingual execution paths and examine their impacts on the test case output.

6.2.1. Bug Overview

Bug2 causes the `Stmt.executeUpdate` method in `sqlite-jdbc 3.7.8` to return an incorrect value. `Stmt.executeUpdate` is a JDBC API method that takes an SQL statement, executes the SQL statement in the database management system, and returns an integer value. This API method returns the number of updated tuples if the input SQL statement is a table-updating statement. This API method has to return zero when the input SQL statement is a table-dropping statement. The cause of the problem is that the call to `NativeDB.changes` native method at Line 187 of `Stmt.java` returns a wrong value (i.e., the previous return value for the most recent table-updating input SQL statement) if the input SQL statement at Line 183 is a table-dropping statement. `NativeDB.changes` returns the number of updated tuples if the last input SQL statement is a table-updating statement.

```
/* Stmt.java */

169: public int executeUpdate(String sql) {
...
183:   statusCode = db._exec(sql) ;
...
187:   changes = db.changes() ;
...
193:   return changes ;
194: }
```

The bug fix inserts two calls to the `NativeDB.total_changes` native method written in C before and after Line 183 and updates `changes` at Line 187 with the difference between the return values of the `total_changes` methods. This bug fix solves the problem because a table-dropping SQL statement does not change the return value of `total_changes` and `executeUpdate` returns zero.

6.2.2. Detailed Experiment Result

We make one failing test case manually based on the bug report and take 149 passing test cases in the regression test suite from `sqlite-jdbc 3.7.8` since the regression test suite has no failing test case.

Table 8 presents the top four most suspicious statements reported by MUSEUM. Lines 187 and 193 of `Stmt.java` tie for the highest suspiciousness rank. These two statements have only one mutant each which sets the return value of the `Stmt.executeUpdate` method as zero. As a result, each of these mutants makes the failing test case pass (and make the two passing test cases fail) and increases the suspiciousness score of the corresponding statement significantly.

Line 183 of `Stmt.java` is ranked as the third most suspicious statement. It has only one mutant that does not execute any SQL query since the `NativeDB._exec` function call is removed by the mutation. This mutant does not fail with the failing test case as `Stmt.executeUpdate` always returns zero when `db._exec(sql)` is not invoked (i.e., the return value of `db.changes` is zero because no SQL query is executed). Meanwhile, the mutant makes 44 passing test cases fail since the mutant does not actually execute the given SQL query. Thus, it increases the second term of the MUSEUM suspiciousness formula.

Line 263 of `Conn.java` is the fourth most suspicious statement together with other 122 statements; it has no mutant that changes the testing result.

6.3. Bug3: Calling a Wrong Native Method in Sqlite-jdbc

Bug3 causes a JDBC API method to return an unexpected value and its cause-effect chain cross Java and C. Bug3 requires testers to follow the flow of data values along multilingual execution paths, which is a challenging and time consuming task.

6.3.1. Bug Overview

Bug3 in `sqlite-jdbc 3.7.15` causes the following erroneous event sequence because the original developers might be confused with the semantics of the `reset` and `clear_binding` native methods in `NativeDB` class:

1. The program creates a `PrepStmt` object in Java, allocates its peer `SQLite` object in C, initializes the peer object with a number of parameters, and executes the peer SQL statement object in C.

Table 8: The four most suspicious statements of Sqlite-jdbc (Bug2)

Rank	Susp. score	Statement	Mutant	$\lvert f(s) \cap p_m \rvert$	$\lvert p(s) \cap f_m \rvert$
2	0.333	`/* Stmt.java:187 */` `changes = db.changes() ;`	`changes = 0 ;`	1	2
2	0.333	`/* Stmt.java:193 */` `return changes ;`	`return 0 ;`	1	2
3	0.332	`/* Stmt.java:183 */` `statusCode = db._exec(sql) ;`	`statusCode = 0 ;`	1	44
126	0.000	`/* Conn.java:263 */` `timeout = ms ;`	`timeout = 0 ;`	0	0

2. The program calls the `PrepStmt.clearParameters` method, which in turn calls the `NativeDB.reset` native method and resets both input query parameters and *output query result*.
3. The program calls a JDBC method (`RS.next`) to obtain the output query result which has an unexpected value.

The bug fix calls the `NativeDB.clear_binding` native method instead of the `NativeDB.reset` native method at Line 64 in the `clearParameters` method because `clear_binding` clears only the input query parameters in the native peer object but `reset` removes both input query parameters and output query results (see the following code):

```
/* PrepStmt.java*/
60:   public void clearParameters() throws
         SQLException {
...
64---    db.reset(pointer);
64+++    db.clear_binding(pointer);
...
```

6.3.2. Detailed Experiment Result

We used 159 test cases which consist of one failing test case in the bug report and 158 passing test cases in the regression test suite. MUSEUM successfully locates Line 64 of `PrepStmt.java` as the most suspicious statement. Table 9 presents the three most suspicious statements and their mutants. The mutant at Line 64 erases the `reset` method call and make both the failing test case and the other 158 passing test cases pass. The next two most suspicious statements (Lines 180 and 131 of `RS.java`) have one partial fix mutant each. These statements have lower suspiciousness scores than Line 64 of `PrepStmt.java` because the partial fix mutants of these statements make 38 and 71 passing test cases fail respectively.

6.4. Bug5: Dangling Pointer to a Native Peer Object in Java-gnome

Bug5 has its effect of a segmentation fault in C while the cause is an attempt to access the freed native peer resource from Java.

6.4.1. Bug Overview

Bug5 crashes JVMs due to a segmentation fault at Line 738 of `gtkspell.c` in Revision 658 of Java-gnome because the `spell` pointer parameter is dangling:

```
/* gtkspell/gtkspell.c */
727: gtkspell_detach(GtkSpell *spell) {
...
738:  g_object_set_data(G_OBJECT(spell->view),
       GTKSPELL_OBJECT_KEY,NULL);
739:  gtkspell_free(spell);
740: }
```

Detailed description of Bug5 is as follows. The `TextView` class of Java-gnome creates a text editor by creating a native peer `GtkTextView` object. The `TextView` class may contain a `Spell` object that provides a spell-checking feature by creating a native peer `GtkSpell` object. In such case, Java-gnome deallocates the `GtkSpell` object by calling `gtkspell_detach` when the corresponding `GtkTextView` object is deallocated. Also, when a `Spell` object is reclaimed, the `Spell.finalize` method calls `Spell.release` method which eventually calls `gtkspell_detach` to deallocate the `GtkSpell` object of the `Spell` object. Thus, a segmentation fault occurs when JVM garbage collector reclaims a `TextView` object (and consequently deallocating `GtkTextView` and `GtkSpell` objects), and then the `Spell` object contained in the `TextView` object.

The bug fix removes Line 57 in the `release` method to avoid the failure:

```
/* Spell.java */
31: public class Spell {
...
56:  protected void release() {
57:   GtkSpell.detach(this) ;
```

Although the fix looks simple, analyzing the buggy statement of Bug5 is challenging because the execution path involves complicated features such as garbage collection, finalization, and reference counting memory management in the external library execution (e.g., glib signal mechanism).

Table 9: Three most suspicious statements of Sqlite-jdbc (Bug3)

Rank	Susp. score	Statement	Mutant	$\|f(s) \cap p_m\|$	$\|p(s) \cap f_m\|$
1	0.1429	`/* PrepStmt.java:64 */` `db.reset(pointer)`	`; // the statement is removed.`	1	0
2	0.0710	`/* RS.java:180 */` `row = 0;`	`row = 1 ;`	1	38
			`; // the statement is removed.`	0	15
3	0.0706	`/* RS.java:131 */` `if (row == 0);`	`if (row != 0)`	1	71
			`if (1)`	0	28

Table 10: Three most suspicious statements of Java-gnome r-695 (Bug5)

Rank	Susp. score	Statement	Mutant	$\|f(s) \cap p_m\|$	$\|p(s) \cap f_m\|$
7	0.020	`/*Spell.java:57*/` `GtkSpell.detach(this);`	`; //the statement is removed.`	1	0
7	0.020	`/*Pointer.java:68*/` `release();`	`; //the statement is removed.`	1	0
7	0.020	`/*Proxy.java:42*/` `super.finalize();`	`; //the statement is removed.`	1	0
7	0.020	`/*GtkSpell.java:48*/` `if (self == null){`	`if(self != null){`	1	0

6.4.2. Detailed Experiment Result

We make one failing test case that reveals Bug5 based on the bug report, and used 183 passing test cases in the Java-gnome regression test suite (revision 659). Our test environment triggers garbage collection at the end of test runs to trigger finalization activities for reclaimed Java objects. To handle the non-deterministic behaviors of garbage collection, we repeat to execute the failing test case 3 times per mutant, and our test oracle reports that a test run fails if at least one out of the three executions with the failing test case fails.

Table 10 presents the four most suspicious statements. Line 57 of `Spell.java` gets the highest suspiciousness score. The mutant at Line 57 is identical to the bug fix. The other six statements have the same suspiciousness score because the mutants of these statements also deactivate `gtkspell_detach` in the Java finalization context. For example, Line 68 of `Pointer.java` and Line 42 of `Proxy.java` (the third and the fourth rows of Table 10) belong to the call sequence from `Spell.finalize` to `gtkspell_detach`; the mutation at Line 48 of `GtkSpell.java` changes the `GtkSpell.detach` method not to call `gtkspell_detach`.

6.5. Bug6: Null Pointer Dereference in Eclipse SWT

Bug6 has a segmentation fault in C while the cause (immature implementation of a callback handler in Eclipse SWT) is in Java.

6.5.1. Bug Overview

Bug 322222 (Bug6) crashes JVMs with a segmentation fault by dereferencing `NULL` at Line 271 of `pango-layout.c`:

```
262: PangoLayout *
263: pango_layout_new (PangoContext *context)
264: {
...
271:    layout->context = context;
...
275: }
```

The origin of `NULL` is the native C function (`callback`) that acts as a gateway from C to Java in the SWT library. `callback` returns `NULL` when a Java exception is pending in the current thread. While the detection of this bug is trivial, debugging Bug6 took a heroic human effort for more than a year with hundreds of comments from dozens of programmers. This bug was difficult for experts to debug since the cause-effect chain goes through Java exception propagation and language transitions. Although the multilingual debuggers [51] aid programmers to locate the origin of `NULL`, they do not locate the buggy statements.

The root cause is an immature implementation of a callback handler at Line 2602 of `Display.java` (the bug report does not describe the root cause but only its symptom, which is often the case for real-world applications. Thus, we had to identify the buggy statement by analyzing the bug patch).

```
// Simplified patch for Bug6
2595  :if(OS.GTK_VERSION>= OS.VERSION(2,4,0)) {
...
2601--:  OS.G_OBJ_CONSTRUCTOR(PLClass);
2602--:  OS.G_OBJ_SET_CONSTRUCTOR(PLClass, newProc);

2601++:  p = OS.G_OBJ_CONSTRUCTOR(PLClass);
2602++:  OS.G_OBJ_SET_CONSTRUCTOR(PLClass,new NewProcCB(p));
```

This patch replaces the `newProc` object that calls `callback` at Line 2602 with a new `NewProcCB(p)` object that calls another callback function that never returns `NULL` in the presence of a pending exception. Although the location of the segmentation fault is far from the callback handler in Java, MUSEUM indicuate the buggy location of the failure accurately (i.e., the suspiciousness rank of Line 2602 is 3 with other two statements).

6.5.2. Detailed Experiment Result

We utilize a test suite consisting of one failing test case and 49 passing test cases. We selected these 49 passing test cases that cover the display module of SWT because all error traces in the bug report contain a method in the display module.

Table 11 presents the four most suspicious statements and their mutants which increase the suspiciousness scores of the corresponding statements (the four statements have only one mutant each). The three most suspicious statements have their ranking as 3 (i.e., MUSEUM reports these three statements as the most suspicious ones) including Line 2602 which causes the failure. The mutants for the three statements change the failing test case into passing one without changing passing test cases (see the fifth and the sixth columns).

These mutants disable the immature callback handler that transitively calls `callback`. The first mutant eliminates Line 2602 that registers the immature callback handler. The second and the third mutants change the return value with zero, which in turn reverses the control flow decision at Line 2595 of `Display.java`, deactivates transitively Line 2602 of registering the immature callback handler, and avoids the segmentation fault. The fourth mutant disables the immature callback handler at the cost of turning one passing test case into a failing one, which decreases the suspiciousness of Line 2392 and lowers the ranking of Line 2392 to 4.

7. Case Study of Debugging Open Bug in Eclipse SWT (Bug8)

While the in-laboratory case studies in Sections 5–6 validate the fault localization accuracy of MUSEUM with respect to the bug reports and their bug fixing revisions, this section demonstrates the usability of MUSEUM in the process of debugging open bugs in real-world open-source software projects. Specifically, our qualitative evaluation demonstrates how to exploit the partial fix mutants from mutation-based analysis as well as the suspicious rankings in diagnosing the exact cause of bugs and suggesting bug fixing patches. Note that an Eclipse maintainer acknowledged our debugging analysis and patch posted at the Eclipse Bugzilla [52].

7.1. Methodology

Bug Description. Bug 419729 (Bug8) in the Eclipse bug repository for Standard Widget Toolkit (SWT) is reported first on October 17, 2013, and it is open and unresolved since we started this case study. We choose this bug for the case study because it appears to be critical for developers and nontrivial to diagnose. First, the "Importance" field of the report is marked as "P3 critical" based on the vote by more than dozens of developers. In addition, the symptom of this bug is a JVM crash, which is obviously undesired in practice. Second, this bug seems to be nontrivial to debug in that this bug had not been resolved for more than 22 months (at the time when we began the case study). Within these 22 months, more than 40 comments from dozens of developers in a few duplicated bug reports did not pinpoint the root cause of this bug nor suggest bug fixing patches.

Bug 419729 is related to the Eclipse SWT module, especially to a subcomponent that binds the SWT interface with the Ubuntu Unity graphics library. Eclipse SWT provides high-level GUI programming interfaces and it uses low-level graphic libraries including Ubuntu Unity to create and manage low-level graphic entities. The subcomponent that binds Eclipse SWT to Ubuntu Unity has native C code as the original interface of Ubuntu Unity is written as C functions.

Participants. Two graduate students among the authors with little background on the target project (i.e., Eclipse SWT and Ubuntu Unity) use debugging tools, diagnose the causes of bugs, produce bug fixing patches, and report their analysis to the bug report database.

Debugging Process. We first run MUSEUM to identify a suspected bug location and obtain a partial fix mutant that makes the failing test case pass. Based on these results, the participants refine the partial fix mutant into a precise and complete patch for the failure.

Debugging Tools. We use the state-of-the-arts research debugging tools including MUSEUM (version 1.3.21) and Blink [19] (version 2.4.0) to locate buggy statements, examine the partial fix mutant, and compare the program states after applying these mutants.

7.2. Debugging the Open Bug Using MUSEUM

Our debugging process consists of fault localization (Section 7.2.1), refining a partial fix mutant (Section 7.2.2), validating the refined mutant (Section 7.2.3), and suggesting a bug patch from the refined mutant (Section 7.2.4).

7.2.1. Fault Localization

Bug 419729 triggers a segmentation fault by dereferencing the `NULL` value in the `state_name` variable at Line 921 of `unity-gtk-action-group.c`. This NULL value is assigned to `state_name` by `unity_gtk_action_group_get_state_name` at Line 920.

Table 11: Four most suspicious statements of the SWT target code (Bug6)

Rank	Susp. score	Statement	Mutant	$\lvert f(s) \cap p_m \rvert$	$\lvert p(s) \cap f_m \rvert$
3	0.0313	`/*Display.java:2602*/ OS.G_OBJ_SET_CONSTRUCTOR (PLClass,NewProc);`	`; /* the function call is removed */`	1	0
3	0.0313	`/*OS.java:8115*/ return _major_version;`	`return 0;`	1	0
3	0.0313	`/*OS.java:8125*/ return _minor_version;`	`return 0;`	1	0
4	0.0306	`/*Display.java:2392*/ initializeSubclasses();`	`; /* the function call is removed */`	1	1

```
/* unity-gtk-action-group.c */
858: void unity_gtk_action_group_connect_item(
       UnityGtkActionGroup *group,
859:   UnityGtkMenuItem *item) {
       ...
920:   state_name =
             unity_gtk_action_group_get_state_name(
                                     group,item);
921:   g_hash_table_insert(action->items_by_name,
                  state_name, g_object_ref(item));
```

To reproduce this bug and localize the buggy statements using MUSEUM, we create one failing test case based on the bug report. Since the original code snippet in the bug report is not a fully self-contained automated test case, we added the following two features to the original code snippet. First, we encoded the user scenario (e.g., mouse-click) in the bug report as automatic GUI events to eliminate human interaction at the test case executions. Second, we made the test case fail when any GUI event in the user scenario is not activated at the test case execution. In addition, we selected 203 passing test cases related to the Eclipse SWT from whole Eclipse regression test suite because the bug report and discussions show that the bug is related to the Ubuntu Unity binding of Eclipse SWT.

Using the one failing and 203 passing test cases, MUSEUM generates 14,490 mutants on the 3,855 out of the 4,998 target source lines covered by the failing test case. Only one mutant makes the failing test case pass (i.e., a partial fix mutant) and the 8,766 mutants make some of the 203 passing test cases fail. MUSEUM generates the partial fix mutant by mutating Line 39339 of `os.c` and reports that line the as most suspicious one:

```
/* os.c */
38334: jlong Java_gtk_radio_menu_item_with_label(...,
                                    jbyteArray arg1) {
         ...
39339:   if ((lparg1=(*env)->GetByteArrayElements(env,
              arg1,NULL))==0)
39340:     goto fail;
39341:   rc = gtk_radio_menu_item_with_label(...,
             lparg1) ;
```

Line 39339 calls the JNI function `GetByteArrayElements` to copy a Java array indicated by `arg1` into a new native array, and the address of the new array is stored in `lparg1`. If the copy operation successes, the address value in `lparg1` flows into `gtk_radio_menu_with_label` as an argument (at Line 39341).

MUSEUM generates the following partial fix mutant at Line 39339 using `Replace-array-elements-with-constants` mutation operator that replaces the `arg1` with a predefined constant byte array `ByteConst`.

```
39339--: if ((lparg1=(*env)->GetByteArrayElements(env,
           arg1, NULL))==0)
39339++: if ((lparg1=(*env)->GetByteArrayElements(env,
           ByteConst, NULL))==0)
```

This mutation changes the flow of values such that the `NULL` value at the failure site (i.e., Line 921 of `unity-gtk-action-group.c`) with the failing test case is replaced with a pointer to a C string derived from `ByteConst`. This mutation does not change the results of the passing test cases.

7.2.2. Refining the Partial Fix Mutant with Failure-inducing Condition

We manually refine the partial fix mutant to make it more accurate by figuring out a *failure-inducing condition* and applying the partial fix only when the condition is true. In other words, we refine the partial mutant to execute the new source line (i.e., 39339++) if the identified failure-inducing condition holds; otherwise, the old source line (i.e., 39339–) is executed.

To identify the failure-inducing condition, we monitored and compared the program states at Line 39339 when running both failing and passing test cases. In the failing executing, the byte array pointed by `lparg1` has its first element as `'\0'` while the first element in the passing executions is not `'\0'`. Thus, we guess that the failure-inducing condition is `lparg1[0]=='\0'`. Using this failure-inducing condition, we refine the partial fix mutant into the following one:

```
if (lparg1[0] == '\0')
  lparg1=(*env)->GetByteArrayElements(env,ByteConst,
                                              NULL);
else
  lparg1=(*env)->GetByteArrayElements(env,arg1,NULL);
```

7.2.3. Validating the Refined Partial Fix Mutant

We validate the refined partial fix mutant by checking if the obtained failure-inducing condition (i.e.,`lparg1[0]=='\0'`) is a general condition to trigger the failure. For that purpose, we compared the execution paths of the original program (i.e., failing execution path) and the refined mutant (i.e., passing execution path) with the same failing test case because we guess that the diversing point between the two execution paths indicates the general condition to trigger the failure. We found that the these executions diverse at Line 766 of `unity-gtk-action-group.c` in the following code snippet:

```
/* unity-gtk-action-group.c */
753: static gchar *
754: unity_gtk_action_group_get_state_name(
       UnityGtkActionGroup *group,
755:   UnityGtkMenuItem    *item) {

756:   gchar  *name = NULL ;
    ...
765:   gchar *label =
         unity_gtk_menu_item_get_label(item) ;
766:   if (label != NULL && label[0] != '\0') {
    ...
800:   else {
    ...}
854:  return name ; }
```

After code review, we found that the then-branch of Line 766 never makes `name` as NULL, which makes `unity_gtk_action_group_get_state_name` return non-NULL-value and avoids the segmentation fault at Line 921. But the else-branch can assign NULL to `name`. The original program execution takes the else-branch while the refined mutant execution takes the then-branch with the failing test case.

As the branch decision at Line 766 depends on the value in `label`, we consider that the value of `label` at Line 766 determines whether the segmentation fault occurs or not. To find which `label` value causes the failure, we monitored the `label` values in the aforementioned two executions (i.e., the executions on the original program and the refined fixing mutant with the failing test case) and the executions with all passing test cases that cover Line 766. The monitoring result shows that, in every test case execution, the array pointed by `label` at Line 766 of `unity-gtk-action-group.c` has the same value as the array pointed by `lparg1` at Line 39339 of `os.c`. For the failing execution, `lparg1[0]` at Line 39339 has `'\0'` value. Meanwhile, in the passing executions, the array pointed by `lparg1` has a non-NULL-value and avoids the crash. Thus, we can conclude that the failure-inducing condition `lparg1[0]=='\0'` is a general condition to trigger the failure and the refined partial fix mutant can fix Bug8.

7.2.4. Suggesting a Bug Fixing Patch

Based on the aforementioned examination of the cause-effect chain, we revised the refined mutant and designed a bug fixing patch. To improve readability, instead of modifying the second argument of `GetByteArrayElement`, we replaced the byte array `lparg1` given to `gtk_radio_menu_item_with_label` with `" "` (a string literal containing one space character) if `lparg1[0] == '\0'`. We posted our analysis on the fault and the following patch to the Eclipse Bugzilla and an Eclipse maintainer acknowledged our analysis and patch [52]:

```
39339: if (lparg1=(*env)->GetByteArrayElements(env,
           arg1,NULL)==0)
39340:   goto fail;
+++    if (lparg1[0] == '\0')
+++      rc=gtk_radio_menu_item_with_label(..., " ");
+++    else
39341:   rc=gtk_radio_menu_item_with_label(...,lparg1);
```

8. Selective Mutation Analyses for Runtime Cost Reduction

8.1. Overview

Although MUSEUM consumes modest amount of time to localize a fault accurately (i.e., 112.6 minutes using 30 machines on average over the eight bugs (Table 3)), we can reduce the runtime cost further at the cost of marginal accuracy loss by carefully selecting mutants and test cases to utilize. Also, by selecting mutants and test cases in various ways, we can control the time cost of fault localization, which is desirable for real-world projects where testing/debugging time budget is tightly given.

We present selective use of mutants and test cases and report the effects of various selection strategies on the accuracy and the cost of fault localization. We have designed total 184 selection strategies based on how to select mutants (23) and how to select test cases (8) and their combinations. For this study, we did not re-execute all target programs and their selected mutants with selected test cases, which would consume unreasonably large amount of time for all 184 selection strategies. Instead, we collected a subset of mutants and a subset of test cases selected by each selection strategy and then compute suspiciousness scores and count the number of mutant executions. If a selection strategy involves randomness, we repeated the selection 30 times to obtain statistical confidences of the result.

Through the study with various selection strategies, we observe that MUSEUM can reduce 96% of the time cost on a single machine for the eight target programs on average (see Table 16) while still locating the buggy statements as the most suspicious statements compared to the full mutation analysis with all mutants and all test cases.

There exist related work that selectively use mutation operators to reduce computational cost of mutation-based fault localization. Papadakis *et al.* [56] presents a mutation-based fault localization tool that uses a small number of mutation operators to avoid heavy cost of mutant executions. Subsequently, Papadakis and Le Traon [57] suggests four sets of selected mutation operators, based on their empirical study of different mutation operator uses and the fault localization results. While the earlier work concentraed on selecting mutation operators, our study explore different chances of selective mutation analyses. For example, our study uses different test case selection criteria and their combinations with mutant selection criteria. In addition, for mutant selection, our studiese includes other criteria than selective uses of mutation operators.

8.2. Selection Strategies

We have studied total 184 (=23×8) selection strategies based on the 23 mutant selection strategies (Section 8.2.1) and the eight test case selection strategies (Section 8.2.2).

8.2.1. Mutant Selection Strategies

We have developed total 23 mutant selection strategies based on the following four criteria where MR(x) and MP(p) are from the existing mutation testing research [58, 59, 60] while MS(n) and MPS(p, n) are developed by the authors:

- **MR(x)**: this strategy randomly selects x% of all generated mutants [58] where $x \in \{10, 20, 30\}$.
- **MS(n)**: it randomly selects n mutants per target line where $n \in \{1, 2, 3\}$. If a target line has only m mutants ($m < n$), MS(n) selects m mutants.
- **MP(p)**: it selects mutants generated by only a certain set of mutation operators p among the following four sets of mutation operators. These four sets consist of the three sets of mutation operators (i.e., SD, CR, and SM) and the set that includes all mutation operators of the three sets:
  - **MP(SD)**: it uses the statement deletion mutation operator [59] together with the 15 new mutation operators for multilingual behavior (Section 3.3).
  - **MP(CR)**: it uses the constant replacement mutation operators [58] together with the 15 new mutation operators for multilingual behavior.
  - **MP(SM)**: it uses the five mutation operators [60] (i.e., 'replace a constant value with its absolute value', 'replace an arithmetic operator with another arithmetic operator', 'change a logical connector', 'change a relational operator', and 'insert an unary operator') with the 15 new mutation operators. Offutt *et al.* [60] claim that mutants generated by these five mutation operators are consistent with the mutants generated by more mutation operators.
  - **MP(All)**: it uses all mutants selected by MP(SD), MP(CR), and MP(SM).
- **MPS(p,n)**: this strategy is a combined strategy of MP(p) and MS(n). Among the mutants selected by MP(p), MPS(p,n) randomly selects n mutants per a target line. If a target line has only m mutants selected by MP(p) ($m < n$), MPS(p,n) randomly selects more mutants generated by other mutation operators to make the target line has n mutants. In this study, we used total 12 strategies by combining p = {SD, CR, SM, All} and $n = \{1, 2, 3\}$.
- **MA**: it selects all generated mutants.

8.2.2. Test Case Selection Strategies

We have developed total eight test case selection strategies based on the random selection and coverage based selection as motivated by the test case selection work [61]. In addition, all test case selection strategies select the failing test case in the test suite.

- **TR(x)**: it randomly selects x% of the passing test cases where $x \in \{10, 20, 30\}$.
- **TC(x)**: it selects x% of the passing test cases that achieve high coverage of the target lines (i.e., the source code lines covered by the failing test case) where $x \in \{10, 20, 30\}$. TC(x) uses a greedy algorithm which repeats to select a passing test case that covers a largest number of uncovered target lines. If there are multiple such passing test cases, the algorithm selects one among the choices.
- **TM**: it selects a small number of passing test cases that cover all target lines. TM uses a greedy algorithm which repeats to select a test case that covers a largest number of uncovered target lines until the selected test cases cover all target lines (the algorithm stops selection when no passing test case can increase the coverage).
- **TA**: TA uses all given passing test cases.

8.2.3. Data Collection

Table 12 shows the results of the mutant selection strategies except MA. Each cell represents the ratio of the number of the selected mutants to the number of all generated mutants. For example, MP(All) selects 2,137 mutants (= 2,861 mutants × 74.7%) for the target code of Bug1 (see the second column of the 11th row of the table).

Table 13 shows the results of the test case selections strategies except TA. Table 13(a) presents the ratio of the

Table 12: Ratio of the number of the selected mutants to the number of all mutants (%)

Strategy	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8	Average
MR(10)	10.0	10.0	10.0	9.8	10.0	9.9	10.0	10.0	10.0
MR(20)	20.0	20.0	19.9	19.8	19.4	20.0	20.0	20.0	19.9
MR(30)	30.0	30.0	29.9	30.1	30.1	30.0	30.1	30.0	30.0
MS(1)	55.5	31.8	32.7	27.1	29.4	25.0	29.7	25.6	32.1
MS(2)	76.9	55.1	56.3	46.8	50.0	44.7	53.6	44.9	53.5
MS(3)	89.2	68.8	69.7	60.1	63.0	60.5	69.1	60.3	67.6
MP(SD)	10.4	22.9	21.2	26.1	28.6	20.0	24.9	22.3	22.1
MP(CR)	56.0	38.5	35.3	19.3	20.9	30.0	35.7	33.6	33.7
MP(SM)	18.5	21.3	20.0	13.7	15.9	11.9	21.5	16.1	17.4
MP(All)	74.7	56.3	53.6	43.1	45.7	51.0	53.5	55.8	54.2
MPS(SD,1)	57.6	41.2	40.8	35.1	39.2	29.5	40.8	33.2	39.7
MPS(SD,2)	77.5	62.3	62.5	52.8	57.4	46.9	61.9	49.4	58.8
MPS(SD,3)	89.3	74.2	74.3	64.2	68.1	61.7	74.8	63.2	71.2
MPS(CR,1)	66.2	49.9	47.6	36.8	40.2	39.1	46.2	41.8	46.0
MPS(CR,2)	79.5	65.7	65.1	52.3	56.7	52.0	63.1	53.4	61.0
MPS(CR,3)	90.3	75.3	74.9	62.9	66.6	63.4	74.5	64.3	71.5
MPS(SM,1)	60.9	40.4	40.3	33.1	36.9	29.7	38.5	33.2	39.1
MPS(SM,2)	78.4	60.7	61.2	50.8	55.1	46.7	59.3	48.7	57.6
MPS(SM,3)	89.8	72.5	72.8	62.9	66.6	61.2	72.5	62.0	70.0
MPS(All,1)	77.6	60.0	57.9	48.1	51.5	51.7	56.6	56.7	57.5
MPS(All,2)	84.9	70.7	69.5	58.1	62.5	58.2	67.5	61.8	66.7
MPS(All,3)	92.7	79.0	77.9	66.4	70.5	66.3	77.9	68.7	74.9

reduced test set size to the original test set size. For example, TM selects 2.3 test cases (=150×1.5%) for Bug2 on average (see the third column of the eighth row of Table 13(a)). TM selects less test cases than TR(x) and TC(x) for all bugs except Bug1 with $x = 10$ or 20 and Bug6 with $x = 10$. Table 13(b) presents the target line coverage achieved by the passing test cases selected by the test case selection strategies. For example, TR(20) covers the 96% of the target lines for Bug1 on average (see the second column of the third row of Table 13(b)). TC(x) achieves the highest target line coverage in all cases except TC(10) on Bug1. TM also achieves the highest coverage with the smallest number of selected test cases among the all strategies that achieve the highest coverage for all target programs (see Table 12(a)). The test case selection strategies do not achieve the 100% coverage if a target line is not covered by any passing test case.

8.3. Effects of the Selection Strategies on Fault Localization

8.3.1. Effect on the Fault Localization Accuracy

Table 14 shows how much the ranking of the faulty line increases with the selection strategies. Table 14(a) presents the increased ranking of the faulty line with the mutant selection strategies (except MA) with all test cases. Table 14(b) presents the increased ranking of the faulty line with the test case selection strategies (except TA) with all mutants. Table 14(c) presents the increased ranking with 12 combined strategies of the four mutant selection strategies (i.e., MPS(p,1) with $p \in \{\text{SD}, \text{CR}, \text{SM}, \text{All}\}$) and the three test case selection strategies (i.e., TR(10), TC(10), and TM). Note that 0 in the table indicates that the ranking of the faulty statement does not change with a given selection strategy (i.e., keeping the same fault localization accuracy).

Table 14(a) shows that all 12 MPS strategies do not increase the ranking of the faulty statement in all target bugs except Bug5, Bug6 and Bug7. Note that even for Bug5, Bug6, and Bug7, MUSEUM still reports the faulty line as the most suspicious one (i.e., MPS increases the number of the most suspicious lines whose suspiciousness scores are all equal to that of the faulty statement). For example, MUSEUM with MPS(SD,1) reports the suspiciousness ranking of the faulty statement in Bug6 as 5.0 (=3+2.0) on average, but still reports the faulty statement as the most suspicious one with other 4.0 statements. However, the other selection strategies in Table 14(a) increase the ranking much. For example, MR and MS increase the ranking much for Bug1, Bug4, Bug7 and Bug8.

Table 14(b) shows that the test case selection strategies with all mutants do not increase the ranking of the faulty statement in all target bugs except Bug2 and Bug5. Even for Bug2 and Bug5, these strategies report the faulty statement as the most suspicious statement with other statements in a tie.

Table 14(c) presents the increased ranking of the faulty statements with the 12 combinations of the four MPS(p,1) strategies where $p \in \{\text{SD}, \text{CR}, \text{SM}, \text{All}\}$ and the three test case selection strategies TR(10), TC(10) and TM. The table shows that these 12 selection strategies increase the ranking by 1.3 on average over all eight target programs. More importantly, these 12 strategies still report the faulty statement as the most suspicious statement with other statements in a tie. For Bug5 and Bug6, the increased

Table 13: Results of the test cases selection strategies

(a) Ratio of the number of the selected test cases to the number of all test cases (%)

Strategy	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8	Average
TR(10)	12.9	10.0	11.0	11.9	11.0	11.0	9.8	10.9	10.1
TR(20)	25.2	19.7	18.9	20.3	22.0	24.7	20.9	20.0	20.1
TR(30)	38.8	29.1	31.3	32.3	32.0	31.6	30.9	30.7	30.0
TC(10)	12.9	10.2	10.6	13.9	12.4	13.3	10.6	11.4	10.1
TC(20)	25.9	20.4	20.6	23.6	21.5	23.0	20.8	21.2	20.1
TC(30)	38.3	30.2	30.5	32.3	31.1	32.5	30.8	30.8	30.0
TM	25.9	1.5	2.4	6.0	3.7	13.3	3.6	7.6	3.3

(b) Target line coverage achieved by the passing test cases selected by the selection strategies (%)

Strategy	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8	Average
TR(10)	0	100	100	55	69	50	93	86	69.1
TR(20)	96	100	98	55	92	81	93	88	87.9
TR(30)	100	100	100	90	94	82	93	95	94.3
TC(10)	0	100	100	90	94	82	93	99	82.3
TC(20)	100	100	100	90	94	82	93	99	94.8
TC(30)	100	100	100	90	94	82	93	99	94.8
TM	100	100	100	90	94	82	93	99	94.8

ranking is larger than the other target programs because the number of mutants that change the test case execution results is reduced significantly for Bug5 and Bug6. For example, the MPS(SD,1) and TM selection strategy decreases the number of mutants that make the failing test case pass from 51 to 26 for Bug5 and from 32 to 7 for Bug6; consequently, more lines have the same numbers of the fail-to-pass mutant executions and pass-to-fail mutant executions as the faulty line after the mutant and test case selections. For the other six mutants, the number of mutants that make the failing test case pass is decreased by 0 to 2. We do not present the results of the other selection strategies because they are worse than these 12 presented strategies. For example, as shown in Table 14(a), MR, MP, and MS degrade the fault localization accuracy significantly. We do not present MPS(p,2) and MPS(p,3) because they are similar to MPS(p,1) in terms of the accuracy but they select much more mutants than MPS(p,1) (Table 12). For the similar reason, we present the results with TR(10), TC(10) and TM, not the other test case selection strategies.

8.3.2. *Effect on the Fault Localization Cost with Marginal Accuracy Loss*

Table 15 presents the ratio of the reduced number of the mutant executions with a selection strategy (i.e., the number of all pairs (m_i, t_{ij}) where m_i is a selected mutant and t_{ij} is a selected test case that covers the mutated line of m_i) to that of the full mutant executions (i.e., mutant executions with all mutants and all test cases). Table 15 shows that the 12 strategies reduce the number of the mutant executions to 3.5%–6.8% of the full mutant executions for the eight target bugs on average. MPS(SD,1) with TM shows the smallest number of mutant executions on average. For example, MPS(SD,1) with TM reduces the number of the mutant executions to 0.6% on Bug3 (i.e., 99.4% reduction of the number of the full mutant executions).

Figure 3 visualizes the relation between the cost reduction (x-axis) and the decreased accuracy (y-axis) of the 12 selection strategies. Each data point corresponds to a selection strategy. The x-value represents the average ratio of the reduced cost of the mutant testing (i.e., the reduced number of the mutant executions) to the cost of the full mutant testing with all mutants and all test cases. The y-value represents the average ranking increase of the faulty statement. For example, MPS(SD,1) with TM reduces the number of the mutant executions to 3.5% of that of the full mutant executions on average (the last column of the fourth row of Table 15) and increases the ranking by 1.4 on average (the last column of the fourth row of Table 14(c)); MPS(SD,1) with TM is represented by '×' located at x=3.5 and y=1.4 in the figure. This figure shows a tendency that more mutant testing achieves higher accuracy. For example, MPS(SD,1) with TC(10) is represented by '×' located at x=4.7 and y=1.1, which indicates that MPS(SD,1) with TC(10) executes more mutant testing than MPS(SD,1) with TM (4.7% v.s. 3.5%) but it increases the ranking less than MPS(SD,1) with TM (1.1 v.s. 1.4). Note that these 12 strategies achieve both high accuracy (i.e., the average ranking increase is less than 2.0) and high cost reduction (i.e., the reduced number of the mutant executions is less than 10% of that of the full mutant executions).

Finally, Table 16 shows the overall time cost of the fault

Table 14: Ranking increase of the faulty statements with various selection strategies
(a) Strategies that reduce the mutants only

Strategy	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8	Average
MR(10)	1,590.7	218.5	340.2	140.3	135.3	2,867.8	207.6	3,760.8	1,157.7
MR(20)	1,571.3	192.0	232.7	97.5	112.6	1,968.3	110.5	3,226.0	938.9
MR(30)	1,368.5	166.8	201.0	62.2	82.6	2,039.8	94.5	2,813.4	853.6
MS(1)	1,141.0	0.0	0.0	103.2	1.9	2.2	118.0	2,257.8	453.0
MS(2)	824.9	0.0	0.0	85.6	0.4	0.4	60.7	1,879.3	356.4
MS(3)	409.3	0.0	0.0	73.8	0.0	0.0	32.0	673.4	148.6
MP(SD)	0.0	238.0	367.0	0.0	-1.0	0.0	0.0	0.0	75.5
MP(CR)	0.0	-1.0	0.0	0.0	153.0	2,960.0	1.0	0.0	389.1
MP(SM)	0.0	244.0	358.0	0.0	162.0	3,213.0	1.0	0.0	497.3
MP(All)	0.0	-1.0	0.0	0.0	1.0	2.0	0.0	0.0	0.3
MPS(SD,1)	0.0	0.0	0.0	0.0	2.7	2.0	0.0	0.0	0.6
MPS(SD,2)	0.0	0.0	0.0	0.0	0.3	1.0	0.0	0.0	0.2
MPS(SD,3)	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.0
MPS(CR,1)	0.0	0.0	0.0	0.0	0.6	3.3	1.0	0.0	0.6
MPS(CR,2)	0.0	0.0	0.0	0.0	0.0	0.9	1.0	0.0	0.2
MPS(CR,3)	0.0	0.0	0.0	0.0	0.0	0.1	1.0	0.0	0.1
MPS(SM,1)	0.0	0.0	0.0	0.0	2.1	2.2	1.0	0.0	0.7
MPS(SM,2)	0.0	0.0	0.0	0.0	0.4	0.7	1.0	0.0	0.3
MPS(SM,3)	0.0	0.0	0.0	0.0	0.0	0.2	1.0	0.0	0.2
MPS(All,1)	0.0	0.0	0.0	0.0	1.0	2.0	0.0	0.0	0.4
MPS(All,2)	0.0	0.0	0.0	0.0	0.0	1.4	0.0	0.0	0.2
MPS(All,3)	0.0	0.0	0.0	0.0	0.0	0.4	0.0	0.0	0.1

(b) Strategies that reduce the test cases only

Strategy	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8	Average
TR(10)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
TR(20)	0.0	0.0	0.0	0.0	3.0	0.0	0.0	0.0	0.4
TR(30)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
TC(10)	0.0	0.0	0.0	0.0	1.8	0.0	0.0	0.0	0.2
TC(20)	0.0	0.0	0.0	0.0	1.6	0.0	0.0	0.0	0.2
TC(30)	0.0	0.0	0.0	0.0	1.1	0.0	0.0	0.0	0.1
TM	0.0	0.4	0.0	0.0	3.0	0.0	0.0	0.0	0.4

(c) Strategies that reduce both mutants and test cases

Strategy		Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8	Average
Mutant	Test case									
MPS(SD,1)	TR(10)	0.0	0.0	0.2	0.0	8.7	2.2	0.3	0.0	1.4
MPS(SD,1)	TC(10)	0.0	0.0	0.0	0.0	6.6	2.1	0.0	0.0	1.1
MPS(SD,1)	TM	0.0	0.5	0.0	0.0	8.5	2.2	0.0	0.0	1.4
MPS(CR,1)	TR(10)	0.0	0.0	0.1	0.0	7.6	3.6	1.0	0.0	1.5
MPS(CR,1)	TC(10)	0.0	0.0	0.0	0.0	4.4	3.3	1.0	0.0	1.1
MPS(CR,1)	TM	0.0	0.4	0.0	0.0	5.0	3.4	1.0	0.0	1.2
MPS(SM,1)	TR(10)	0.0	0.1	0.0	0.0	9.2	1.9	1.0	0.0	1.5
MPS(SM,1)	TC(10)	0.0	0.0	0.0	0.0	6.6	1.9	1.0	0.0	1.2
MPS(SM,1)	TM	0.0	0.4	0.0	0.0	7.8	1.9	1.0	0.0	1.4
MPS(All,1)	TR(10)	0.0	0.0	0.2	0.0	7.6	2.0	0.0	0.0	1.2
MPS(All,1)	TC(10)	0.0	0.0	0.0	0.0	5.0	2.0	0.0	0.0	0.9
MPS(All,1)	TM	0.0	0.3	0.0	0.0	6.0	2.0	0.0	0.0	1.0

localization with all mutants and all test cases (the second row) and that of the fault localization with MPS(SD,1) and TM (the third row) on one machine. The numbers in the second row are calculated by multiplying 30 to the time cost in Table 3. MUSEUM with the MPS(SD,1) and TM selection strategies consumes only 3.8% of the time cost with all mutants and all test cases for the eight target bugs on average (see the last column of the last row). Thus, this result confirms that the selection strategy can effectively reduce the time cost of MUSEUM as the number of the mutant executions is reduced. [7]

[7]The ratio in Table 16 can be different from the ratio in Table 12 because the time cost of MUSEUM involves mutant generations, data processing and other operational steps in addition to mutant executions (also execution time of a mutant can be different depending on the mutant and the test case used).

Table 15: The ratio of the reduced number of the mutant executions to that of the full mutant executions (%)

Strategy Mutant	Test case	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8	Average
MPS(SD,1)	TR(10)	7.4	4.2	4.2	3.9	4.4	3.6	4.3	3.6	4.5
MPS(SD,1)	TC(10)	7.4	4.1	4.3	4.4	4.7	4.2	4.4	3.8	4.7
MPS(SD,1)	TM	14.8	0.6	1.0	1.6	1.4	4.2	1.5	2.5	3.5
MPS(CR,1)	TR(10)	8.5	5.0	4.9	4.3	4.5	4.6	4.8	4.5	5.1
MPS(CR,1)	TC(10)	8.5	5.0	5.0	4.8	4.8	5.3	5.0	4.8	5.4
MPS(CR,1)	TM	17.0	0.7	1.1	1.8	1.4	5.3	1.6	3.2	4.0
MPS(SM,1)	TR(10)	7.8	4.1	4.1	3.8	4.2	3.6	4.0	3.6	4.4
MPS(SM,1)	TC(10)	7.8	4.2	4.1	4.1	4.5	4.2	4.2	3.8	4.6
MPS(SM,1)	TM	15.7	0.6	1.0	1.5	1.3	4.2	1.4	2.6	3.5
MPS(All,1)	TR(10)	10.0	6.0	5.8	5.6	5.7	6.2	5.9	6.1	6.4
MPS(All,1)	TC(10)	10.0	6.0	6.1	6.3	6.3	7.1	6.0	6.6	6.8
MPS(All,1)	TM	20.0	0.9	1.4	2.5	1.8	7.1	2.0	4.4	6.5

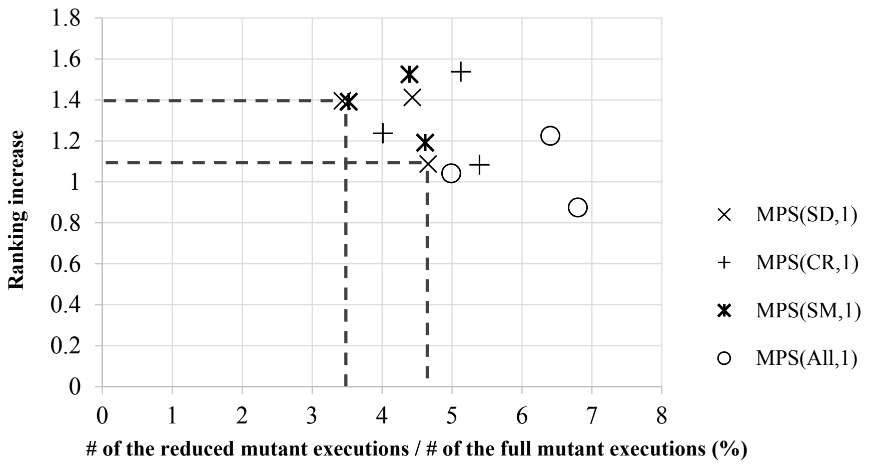

Figure 3: Rank increase and the ratio of the reduced number of mutant executions to that of the full mutant executions

9. Discussions

9.1. *Advantages of the Mutation-based Fault Localization for Real-world Multilingual Programs*

One of the reasons that make debugging real-world programs difficult is the poor quality of a test suite because fault localization can be more accurate if a test suite covers more diverse execution paths. For large real-world programs, it is challenging to build test cases that exercise diverse execution paths because it is non-trivial to understand and control a target program. In addition, generating diverse test cases for multilingual programs has additional burden to learn and satisfy safety rules on language interface such as JNI constraints. Thus, multilingual programs are often developed with only simple test cases. As a result, as shown in Table 4, the SBFL techniques fail to accurately localize the eight real-world multilingual programs.

For example, the statement coverages of the test suites used for Bug2 and Bug3 are around 85% and 86% and the SBFL techniques localize these bugs somehow precisely (i.e., the suspiciousness rank of Bug2 and Bug3 are 4 and 5, respectively). However, the statement coverages of the test suites used for Bug1, Bug4, Bug5, Bug6, and Bug8 are around 1%, 22%, 24%, 19%, and 11% and the accuracy of the SBFL techniques for these bugs are very low (Table 4). In contrast, MUSEUM can alleviate this limitation by achieving the effect of diverse test cases through the diverse mutants with limited test cases. Thus, MUSEUM can be a promising technique for debugging complex real-world multilingual programs.

9.2. *Effectiveness of the New Mutation Operators for Localizing Multilingual Bugs*

Table 17 presents the information on the mutation operators that generate mutants on which the failing test case passes (i.e., partial fix mutants). The second and the third rows present the number of all mutation operators and that of the new mutation operators for multilingual behaviors (Section 3.3) that generate partial fixes, respectively. Table 17 shows that the new mutation operators are effective to generate informative mutants (i.e., partial fix mutants) to localize multilingual bugs. In other words, the table shows that only the new mutation operators generate partial fix mutants for Bug1, Bug4 and Bug8 (i.e., since the numbers in the second row and the third row are the same). For Bug1, only the `Pin-Java-Object` mutation operator generates the partial

Table 16: Overall time cost of fault localization (in minutes)

	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8	Average
MUSEUM with all mutants and all test cases	360	1,785	1,346	738	682	5,262	1,501	15,334	3,376.0
MUSEUM with MPS(SD,1) and TM	29	21	34	10	10	186	63	1,166	189.9
Ratio	8.1%	1.2%	2.5%	1.4%	1.5%	3.5%	4.2%	7.6%	3.8%

Table 17: Statistics on the mutation operators that generate mutants in the experiments

	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8
# of mutation operators that generate a partial fix mutant	1	3	6	2	12	14	2	1
# of the *new* mutation operators that generate a partial fix mutant	1	0	0	2	1	0	1	1

fix mutant. For Bug4, only `Make-global-reference` and `Make-weak-global-reference` generate the partial fix mutants. Similarly, for Bug8, only `Replace-array-elements-with-constants` generates one.

To assess the impact of the new mutation operators on fault localization, we ran MUSEUM for Bug1, Bug4, Bug5, Bug7 and Bug8 (which have partial fix mutants by the new mutation operators) without the new mutation operators. For Bug1, Bug4 and Bug8 which have partial fix mutants generated by only the new mutation operators, the suspiciousness ranking of the faulty line becomes significantly low (1737 for Bug1 (89.6%), 117 (62.9%) for Bug4, and 3061 (61.2%) for Bug8). For Bug5 which has also the partial fix mutants generated by the conventional mutation operators, the ranking of the faulty line changes from the eighth to the ninth and the faulty line is not anymore the most suspicious statement. For Bug7, the ranking of the faulty line remains unchanged.

9.3. High Accuracy with Low Runtime Cost through Selective Mutation Analysis

The result of the selective mutation analysis shows that MUSEUM can achieve high fault localization accuracy with significantly reduced runtime cost compared to that of the full mutant executions (e.g., MPS(SD,1) and TM can reduce the runtime cost up to 96% and identifies the faulty statements as the most suspicious ones) (Section 8.3.1). Also, we observe that there is a tendency that more mutants and test cases can increase the fault localization accuracy with the selective mutation analysis (Figure 3).

A practical implication from these observations is as follows. MUSEUM should start with the mutants and test cases selected by a selection strategy that reduces the runtime cost in a large degree but decreases the fault localization accuracy marginally, such as MPS(SD,1) and TM. Then, MUSEUM can add more mutants and test cases by gradually relaxing the parameter of the selection strategy (e.g, selecting the test cases of TC(10) to TC(100)) or changing the selection strategy. MUSEUM generates additional mutant executions of the added mutants and the added test cases and re-calculates the suspiciousness scores based on the updated mutant testing result, which consumes little computing power. In this way, MUSEUM can achieve high fault localization accuracy with the small amount of the runtime cost first and then increase the fault localization accuracy gradually within the given time budget.

10. Conclusion and Future Work

We have presented MUSEUM which localizes bugs in complex real-world multilingual programs in a language agnostic manner through mutation analyses. The experiments on the eight real-world multilingual programs show that MUSEUM precisely locates the faulty statement for all non-trivial Java/C bugs. In addition, we show that the accuracy of fault localization for multilingual programs can be increased by adding new mutation operators relevant with language interface constraints.

As future work, we will add more mutation operators targeting features in multilingual programs and modify existing mutation operators to reduce equivalent mutants. Also, we will apply MUSEUM to an interactive debugger such as Blink [51] and/or advanced automated testing techniques [62, 63] to maximize the debugging effectiveness. Finally, we will investigate effective methods to utilize MUSEUM to improve automated program repair and/or search-based program analysis for multilingual programs.

Acknowledgement

This work is supported by the ITRC support program (IITP-2015-H8501-15-1012), and the NRF grants (NRF-2014K1A3A1A09063167, NRF-2015R1C1A1A01055259, and

NRF-2015R1C1A1A01052876) funded by the Korea government (MSIP), and the IITP grant programs funded by the MSIP (Research and Development of Dual Operating System Architecture with High-Reliable RTOS and High-Performance OS [No. R0101-15-0081]; High Performance Big Data Analytics Platform Performance Acceleration Technologies Development [No. R0190-15-2012]).

References

[1] Meyerovich LA, Rabkin AS. Empirical analysis of programming language adoption. *ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2013.

[2] Lee B, Wiedermann B, Hirzel M, Grimm R, McKinley K. Jinn: Synthesizing dynamic bug detectors for foreign language interfaces. *ACM Conference on Program Language Design and Implementation (PLDI)*, 2000.

[3] Liang S. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, 1999.

[4] Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing. *ACM Conference on Program Language Design and Implementation (PLDI)*, 2005.

[5] Kim M, Kim Y, Choi Y. Concolic testing of the multi-sector read operation for flash storage platform software. *Formal Aspects of Computing (FAC)* 2012; **24**(2):355–374.

[6] Kim M, Kim Y, Kim Y. Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE. *International Conference on Software Engineering (ICSE) Software Engineering in Practice track*, 2012.

[7] Kim Y, Kim Y, Kim T, Lee G, Jang Y, Kim M. Automated unit testing of large industrial embedded software using concolic testing. *IEEE/ACM Automated Software Engineering (ASE) Experience track*, 2013.

[8] Park Y, Hong S, Kim M, Lee D, Cho J, Kim M, Kim Y, Kim Y. Systematic testing of reactive software with non-deterministic events: A case study on LG electric oven. *International Conference on Software Engineering (ICSE) Software Engineering in Practice track*, 2015.

[9] Kondoh G, Onodera T. Finding bugs in Java Native Interface programs. *International Symposium on Software Testing and Analysis (ISSTA)*, 2008.

[10] Li S, Tan G. Finding bugs in exceptional situations of JNI programs. *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[11] Li S, Tan G. Finding reference-counting errors in Python/C programs with affine analysis. *European Conference on Object-Oriented Programming (ECOOP)*, 2014.

[12] Li S, Tan G. JET: exception checking in the java native interface. *ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, 2011.

[13] Ravitch T, Jackson S, Aderhold E, Liblit B. Automatic generation of library bindings using static analysis. *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[14] Ravitch T, Liblit B. Analyzing memory ownership patterns in C libraries. *International Symposium on Memory Management (ISMM)*, 2013.

[15] Siefers J, Tan G, Morrisett G. Robusta: taming the native beast of the JVM. *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[16] Tan G, Morrisett G. Ilea: inter-language analysis across Java and C. *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

[17] Moon S, Kim Y, Kim M, Yoo S. Ask the mutants: Mutating faulty programs for fault localization. *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2014.

[18] Hong S, Lee B, Kwak T, Jeon Y, Ko B, Kim Y, Kim M. Mutation-based fault localization for real-world multilingual programs. *IEEE/ACM International Conference on. Automated Software Engineering (ASE)*, 2015.

[19] Lee B, Hirzel M, Grimm R, McKinley KS. Debug all your code: Portable mixed-environment debugging. *ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2009.

[20] MAAlipour. Automated fault localization techniques: a survey. *Technical Report*, Oregon State University 2012.

[21] Wong E, Debroy V. A survey of software fault localization. *Technical Report UTDCS-45-09*, University of Texas at Dallas 2009.

[22] Jones JA, Harrold MJ. Empirical evaluation of the Tarantula automatic fault-localization technique. *IEEE/ACM International Conference on. Automated Software Engineering (ASE)*, 2005.

[23] RAbreu, PZoeteweij, Gemund A. An evaluation of similarity coefficients for software fault localization. *Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2006.

[24] Xie X, Chen TY, Kuo FC, Xu B. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2013; **22**(4):31:1–31:40.

[25] Yoo S, Xie X, Kuo FC, Chen TY, Harman M. No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist. RN/14/14, Department of Computer Science, University College London 2014.

[26] Zhang L, Zhang L, Khurshid S. Injecting mechanical faults to localize developer faults for evolving software. *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

[27] Papadakis M, Le-Traon Y. Using mutants to locate "unknown" faults. *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Mutation Workshop, 2012.

[28] Papadakis M, Le-Traon Y. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 2015; **25**:605–628.

[29] Arnold M, Vechev M, Yahav E. QVM: An efficient runtime for detecting defects in deployed systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2011; **21**(1):2:1–2:35.

[30] Jung C, Lee S, Raman E, Pande S. Automated memory leak detection for production use. *International Conference on Software Engineering (ICSE)*, 2014.

[31] Xu G, Rountev A. Precise memory leak detection for java software using container profiling. *International Conference on Software Engineering (ICSE)*, 2008.

[32] Clause J, Orso A. LEAKPOINT: Pinpointing the causes of memory leaks. *International Conference on Software Engineering (ICSE)*, 2010.

[33] Xu G, Bond MD, Qin F, Rountev A. LeakChaser: Helping programmers narrow down causes of memory leaks. *ACM Conference on Program Language Design and Implementation (PLDI)*, 2011.

[34] Agrawal H, DeMillo RA, Hathaway B, Hsu W, Hsu W, Krauser EW, Martin RJ, Mathur AP, Spafford E. Design of mutant operators for the C programming language. *Technical Report SERC-TR-120-P*, Purdue University 1989.

[35] JNI APIs and developer guides. `http://docs.oracle.com/javase/8/docs/technotes/guides/jni` 2015.

[36] Tan G, Croft J. An empirical security study of the native code in the JDK. *USENIX Security Symposium (SS)*, 2008.

[37] Furr M, Foster JS. Checking type safety of foreign function calls. *ACM Transactions on Programming Languages and Systems* 2008; **30**(4):1–63.

[38] Dawson M, Johnson G, Low A. Best practices for using the Java Native Interface. IBM developerWorks 2009.

[39] JNI Local Reference Changes in ICS. Android Developers Blog. `http://android-developers.blogspot.com/2011/11/jni-local-reference-changes-in-ics.html` 2011.

[40] Tan G, Appel A, Chakradhar S, Raghunathan A, Ravi S, Wang

D. Safe Java Native Interface. *IEEE International Symposium on Secure Software Engineering (ISSSE)*, 2006.

[41] Firefox Bug 958706 - Dont́ hide JNI exceptions. https://bugzilla.mozilla.org/show_bug.cgi?id=958706 2014.

[42] JDK-4804447: JNI get⟨type⟩arrayelements fail with zero length arrays. https://bugs.openjdk.java.net/browse/JDK-4804447 2003.

[43] PIT v0.33 - mutation testing tool for Java. http://pitest.org.

[44] Maldonado JC, Delamaro ME, Fabbri SC, da Silva Simão A, Sugeta T, Vincenzi AMR, Masiero PC. Proteum: A family of tools to support specification and program testing based on mutation. *Mutation testing for the new century*. Kluwer Academic Publishers, 2001.

[45] clang: a c language family fronted for LLVM. http://clang.llvm.org/.

[46] Bruneton E, Lenglet R, Coupaye T. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and Extensible Component Systems* 2002; **30**.

[47] Azureus-commitlog: ListView.java. http://sourceforge.net/p/azureus/mailman/message/18318135/ 2008.

[48] Xerial SQLite-JDBC, Issue 16: DDL statements return result other than 0. https://bitbucket.org/xerial/sqlite-jdbc/issue/16 2012.

[49] Xerial SQLite-JDBC, Issue 36: Calling PreparedStatement.clearParameters() after a ResultSet is opened, causes subsequent calls to the ResultSet to return null. https://bitbucket.org/xerial/sqlite-jdbc/issue/36 2013.

[50] Java-GNOME Avoid segfault lurking in GtkSpell library. https://openhub.net/p/java-gnome-gstreamer/commits/167384488 2009.

[51] Lee B, Hirzel M, Grimm R, McKinley KS. Debugging mixed-environment programs with blink. *Software: Practice and Experience* 2014; **45**(9):1277–1306.

[52] Eclipse SWT bug419729: Native crash in org.eclipse.swt.internal.gtk.OS._gtk_widget_show. https://bugs.eclipse.org/bugs/show_bug.cgi?id=419729 2015.

[53] Jaccard P. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bull. Soc. vaud. Sci. nat* 1901; **37**:547–579.

[54] Ochiai A. Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions. *Bull. Jpn. Soc. Sci. Fish.* 1957; **22**(9):526–530.

[55] Naish L, Lee HJ, Ramamohanarao K. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* August 2011; **20**(3):11:1–11:32.

[56] Papadakis M, Delamaro ME, Traon YL. Proteum/FL: a tool for localizing faults using mutation analysis. *Proceeding of IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2013.

[57] Papadakis M, Traon YL. Effective fault localization via mutation analysis: Selective mutation approach. *Proceedings of ACM Symposium on Applied Computing (SAC)*, 2014.

[58] Ammann P, Delamaro ME, Offutt J. Establishing theoretical minimal sets of mutants. *IEEE International Conference on Software Testing, Verification, and Validation (ICST)*, 2014.

[59] Deng L, Offutt J, Li N. Empirical evaluation of the statement deletion mutation operator. *International Conference on Software Testing, Verification, and Validation (ICST)*, 2013.

[60] Offutt J, Rothermel G, Zapf C. An experimental evaluation of selective mutation. *International Conference on Software Engineering (ICSE)*, 1993.

[61] Graves T, Harrold MJ, Kim JM, Porter A, Rothermel G. An empirical study of regression test selection techniques. *Proceedings of the 20th International Conference on Software Engineering (ICSE)*, 1998.

[62] Hong S, Ahn J, Park S, Kim M, Harrold MJ. Testing concurrent programs to achieve high synchronization coverage. *International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[63] Kim Y, Xu Z, Kim M, Cohen M, Rothermel G. Hybrid directed test suite augmentation: An interleaving framework. *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2014.