Automated unit-level testing of java memory leaks

Lijuan Hong*, Ju Qian, Jifeng Cui

College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China

Received 15 December 2013; Accepted 29 August 2014, www.cmnt.lv

Abstract

Java programs may suffer from serious memory leak bugs. To resolve these bugs, various leak diagnosing and even fixing techniques have been proposed. However, in literature, there is very few work, which focuses on memory leak testing. Without revealing leak phenomenon by testing in advance, even excellent leak diagnosing and fixing techniques can be hard to work. In software testing, unit testing is a technique to avoid faults at early stage of software development. This paper proposes an automated unit-level memory leak testing approach to find potential leak bugs in Java methods. It firstly identifies the methods with high leaking risks. Then, strengthened unit tests are generated accordingly to check whether those risky modules can really cause leaks. Cases studies show that our method could be valuable for real programs.

Keywords: Java, memory leak, unit testing, test generation

1 Introduction

Even though with garbage collection supports, memory leak still remains a problem for Java programs. The leaks usually occur when a Java program unnecessarily maintains references to objects that are no longer required. Memory Leaks may degrade runtime performance and even lead to crashes due to out of memory exceptions.

To resolve these leak bugs, various techniques have been proposed [1-7], and there are also a lot of supporting tools [11, 12]. The previous work mainly focuses on leak diagnosing and fixing, which find out the causes of leaks after memory leak phenomenon occur [1-7] and fix the leaking code [8-10]. However, in literature, there is very few work concerning on how to discover those potential memory leak phenomenon. Without a discovered leak phenomena, in most cases, it will be hard to trigger a diagnosis process and eliminate the leak error.

Software testing is a promising technique to discover potential failures. But approaches for testing memory leaks are hard to see. In this paper, we present an automated testing method to find memory leak phenomenon at unit level. The approach firstly finds out the leak risky methods and then generates normal unit tests for them. We then strengthen these unit tests to detect memory leaks. By testing leaks at unit-level, memory leaks can be avoided as early as possible in the whole lifecycle of software.

In the work, we classify the leak risky modules into three categories: modules directly creating and leaking large number of objects, modules that accumulate new objects created by themselves and may lead to leaks after repeated calls to them, modules that absorb their arguments and may lead to leaks after repeated calls. These modules can be identified with dependency, points-to, and escape information.

For those leak risky modules, we firstly use some existing approaches [13] to generate normal unit tests as start points. Then, these tests are augmented with large input data, repeated calls, and other techniques to strengthen the memory usage and monitor mechanisms to observe the leak. Finally, we can get the leak revealing unit tests. We studied usage of our approach on several already found memory leak bugs from JDK bug database. The results show that the approach is effective in revealing real leaking modules. This indicates that it can be valuable for practical uses.

2 Technique backgrounds

2.1 DEPENDENCE ANALYSIS

Dependency between program statements can be categorized into two types: control dependence caused by control structures in the program and data dependence caused by reads and writes of memory locations [14].

We can get control and data dependences between nodes in control flow graph via dependence analysis [14]. In this paper, we need to know whether a loop's control condition is influenced by method parameters. By dependence analysis, we can get the data dependence relationships between program nodes, and then judge whether the loop's condition node directly or indirectly depends on method parameter by checking whether there is a path from the method parameter to the condition node. If there exist such kinds of paths, it indicates that the loop's control condition is potentially influenced by method's parameters.

2.2 POINTER AND ESCAPE ANALYSIS

Pointer analysis determines all the possible memory locations that a pointer may point to at runtime. In Java, a pointer is a variable of reference type and what it points to is an object on the heap [15].

This paper uses a context-insensitive algorithm implemented in Soot [17] to do the pointer analysis.

*Corresponding author e-mail ljhong307@163.com

Escape analysis tracks the lifetime of objects and determine whether it may escape from some given scopes. An object can be directly created by a new instruction or indirectly created by a wrapper method of some new instruction. It is considered to possibly escape from the scope of a method if a reference to the object is returned from the method, or if a reference to the object is assigned to a field of an external object [16].

This paper extends the escape analysis presented in [16] to do risky method analysis. In addition to that in [16], the extended analysis also adds loops as the analysed scopes. We firstly identify all objects created in the loops. Then, a constraint system is built according to the statements in the relevant method following the constraint-based approach in [16]. Finally, we can determine whether the objects in the loop may escape loop scope by the extended escape analysis.

3 Leak risky modules: a classification

As discussed in the introduction part, the leak risky modules can be classified into three categories. We will introduce them in more detail in this section.

3.1 MODULES DIRECTLY LEAKING OBJECTS

Modules directly creating and leaking objects have the following characteristics:

1) Creating objects repeatedly in loops inside the module;

2) The number of rounds that these loops can execute is unbounded and determined by the module inputs;

3) The objects created by the loops cannot be released on time. Such modules may directly consume large memory when the inputs are large. The memory requirement may be caused by improper object allocation and release mechanism and is unnecessary. It may cause memory leaks.

```
class Test1{
1    static Vector cache = new Vector();
2    public void foo(int n){
3       for( int i = 1; i < n; i ++ ){
4          Data d = new Data();
5          cache.add(d);
6          doSth(d);
7       }
8    }
}
```

FIGURE 1 An example of modules directly leaking objects

Figure 1 demonstrates an example for the modules that directly creating and leaking objects. In Figure 1, method foo may directly cause memory leaks. In foo, there is a loop which creates new objects inside it. The execution rounds of the loop are unbounded and determined by the method's input parameter *n*. During each round, object d created in the loop is added into an external cache. The cached objects are not freed on time. Given a very large input, the method may directly consume too much memory and lead to out of memory error.

3.2 MODULES ACCUMULATING NEWLY CREATED OBJECTS

Modules that accumulate newly created objects and can lead to memory leak usually have the following characteristics:

1) Creating objects inside the module;

2) The objects escape from module's scope and get stored through a way other than the method return value and parameters. It may unconsciously consume large memory after repeated calls and thereby cause memory leaks.

```
class Test2{
1    static Vector cache = new Vector();
2    public Data bar(){
3       Data d = new Data();
4       cache.add(d);
5       return d;
6    }
  }
```

FIGURE 2 An example of modules accumulating newly created objects

Figure 2 demonstrates an example for the modules of the second category. In Figure 2, method bar creates a new object d and accumulates it into container cache. If the cache is not cleaned on time, after a large number of calls to the bar() method, there will be too many Data objects stored in the container cache, which may cause memory leaks.

3.3 MODULES ABSORBING ARGUMENT OBJECTS

A module that absorbs argument objects and can lead to memory leak usually has at least one of its parameter objects potentially escaping out of the module's scope through a way other than the method return value and parameters. The escaped parameter object can be long-termly absorbed by the module. It may lead to memory leaks after many calls to the module.

```
class Test3{
1    static Vector cache = new Vector();
2    public void zar(Data d){
3       cache.add(d);
4       doSth(d);
5    }
  }
```

FIGURE 3 An example of modules absorbing argument objects

Figure 3 shows an example for the modules of this category. In Figure 3, a reference type parameter is passed into method zar in line 2. In line 3, parameter d is absorbed by method zar to an external cache. The method may unconsciously absorb too many parameter objects after repeated calls, which may causes leaks and finally lead to out of memory error.

4 Identifying Leak Risky Modules

This section presents the methods for identifying three kinds of leak risky modules, respectively.

4.1 IDENTIFYING MODULES DIRECTLY LEAKING OBJECTS

We firstly search the loops that create new objects in a method on the control flow graph. Then whether these loops' control conditions can be influenced by the method inputs are determined. Finally, we use pointer and escape analyses to determine whether the new objects' created in those loops can live beyond the loop scope. If the condition of a loop that creates new objects is potentially influenced by the method inputs and the created objects can live beyond the loop scope, it indicates that the module has a high risk in directly leaking huge memory.

Algorithm 1: Identifying modules that directly leaking objects

```
Input: m: Method
Output: Boolean
   Dependence analysis;
pointer and escape analyses;
let L_m be the set of all loops in m
foreach l∈L_m do
     if hasNewInstruction(l) then
      if isInputDependent(l) then
if isNewObjectEscape(l) then
return true;
end
        end
    end
  end
return false;
```

The algorithm is shown in Algorithm 1. It returns a Boolean value for each input method m to show whether the method may cause leaks. The algorithm firstly collects Lm, the set of all loops in m, by loop analysis. Then, each loop l is processed. We firstly check whether loop l can introduce new objects by predicate hasNewInstruction(l). The new objects include the ones created directly by new instructions and the ones created by other callee methods. Then we determine whether loop l is input dependent on the method's parameters by predicate isInputDependent(l). Finally, we use predicate isNewObjectEscape(l) to check whether the newly introduced objects may escape from the loop scope. When all the above conditions are satisfied, it indicates that method m is a leak risky module.

For isInputDependent(l), we firstly obtain the dependence node corresponding to the loop condition. Starting from this node, we traverse the program dependence graph to get a set of nodes that the loop condition node depends on. If the set contains any node corresponding to the method's parameter, it indicates loop l's condition depends on method inputs.

For isNewObjectEscape(l), we firstly check whether the objects created in the loop escape from the method scope by existing escape analysis. If they escape, of course the objects escape from the loop scope. Otherwise, we will check whether the objects created by the loop may escape from the loop scope by our extended escape analysis.

4.2 IDENTIFYING MODULES ACCUMULATING NEWLY CREATED OBJECTS

We firstly obtain all the objects newly created in a module. Then, pointer and escape analysis are used to determine lifetime of these objects and whether they may escape from the method scope via internal leak sources and thereby be accumulated.

The algorithm is shown in Algorithm 2. It returns a Boolean value for each input method m to show whether the method may cause leaks after repeated calls. We firstly obtain a collection of internal escape sources excluding the return value and parameters by function getInternalLeakSources(). If method m accumulates its created objects, the new objects will escape from these sources. With these special escape sources, we do escape analysis for the method. Having got the escape information, we check each newly created object in the method, and finally determine whether there is any object escaping from the method scope by predicate isEscape(o). If such objects exist, the method may potentially cause memory leaks.

Algorithm 2: Identifying modules that accumulating newly created objects

```
Input: m:method
Output: Boolean
   escape_sources := getInternalLeakSources();
   pointer and escape analysis;
   newObjects := getAllNewObjects();
   foreach o∈newObjects do
     if isEscape(o) then
        return true;
    end
end
return false;
```

4.3 IDENTIFYING MODULES ABSORBING ARGUMENT OBJECTS

We firstly check whether a module's parameters are reference types and regard the reference typed parameters as newly created objects in the modules. Then, pointer and escape analyses can be used to find out the lifetime of these objects and determine whether they may escape from method scope.

Algorithm 3: Identifying modules that absorbing argument objects

```
Input: m:method
Output: Boolean
     newObjects := markParamsAsNewObject (m);
pointer and escape analyses;
foreach o∈newObjects then
if isEscape(o) then
return true;
          end
    end
return false;
```

The algorithm is shown in Algorithm 3. It is similar to Algorithm 2. We firstly set method's reference parameters as newly created objects by markParamsAsNewObject(m). With these special newObjects, we do pointer and escape analyses for the method. Based on the escape information,

we check each newly created object in the method to determine whether there is any object escaping from method scope by predicate isEscape(o). If such objects exist, the method is a leak risky module.

5 Creating unit tests

For the leak risky modules, the users can pick up the modules that they believe should not cause continuously memory growth to do the unit testing.

Our unit test generation approach firstly uses some existing methods to generate normal unit tests. Then, we use enlarged input data, repeated method calls, and other techniques to strengthen their memory use, and insert memory monitor mechanisms to observe the leaking behaviors. Finally a collection of unit tests for memory leak testing purpose can be obtained.

The approach creates JUnit format test cases. In the current implementation, we use a tool named CodeProAnalytix [13] to generate normal JUnit test cases for the risky modules. The augmenting methods are discussed below.

5.1 CREATING UNIT TESTS FOR MODULES DIRECTLY LEAKING OBJECTS

For the modules that could directly cause leaks, we use large input data to strengthen memory use and insert memory growth assertions to determine whether the leaks can really occur.

```
public class UnitTest1 {
1       @Before
2       public void setUp() throws Exception {
3       }
4       @Test
5       public void testFoo(){
6           Test1 test = new Test1();
7           int n = MemoryTester.LARGE_INT;
8           long memoryBefore = MemoryTester.getUsed-
Memory();
9           test.foo(n);
10                  long memoryAfter =
MemoryTester.getUsedMemory();
11      MemoryTester.assertMemoryGrowth(memoryBefore,
memoryAfter, NO_SIGNIFICANT_GROWTH);
12  }
}
```

FIGURE 4 Unit testing with large input data and memory assertions

Figure 4 shows a unit test generated for the example in Figure 1. The normal unit test generated by the existing tools only contains the creation of Test1 object and a call to its method foo. In the normal unit test, it passes a random initial value to the tested method. To test memory leaks, in statement 7, we set a large input data MemoryTester.LARGE_INT for the tested method foo to strengthen memory use. Our approach currently supports several different types of large data, including the primitive types, such as int, long, float, and so on, and some object types, such as String. For the primitive types, we just use some previously defined huge value. For String type, we generate a pool of large strings and randomly select one of them. We obtain the memory consummation before and after foo by calls to method MemoryTester.getUsedMemory() and determine whether the method cause leaks by assertion MemoryTester.assertMemoryGrowth(memoryBefore, memoryAfter, NO_SIGNIFICANT_GROWTH). The assertion checks whether the memory growth is in the normal range. It uses a predefined value NO_SIGNIFICANT_GROWTH to set the allowed growth range. The constant means only small memory growth is allowed. If the memory grows over the allowed value, we consider the risky method really causes memory leaks. With the above strengthen, the unit test can validate whether the leak risky method actually lead to a noticeable leak.

5.2 CREATING UNIT TESTS FOR MODULES ACCUMULATING NEWLY CREATED OBJECTS

For the second kind of leak risky modules, we use repeated method calls to strengthen the memory usage, and insert memory growth assertions to check whether the leak risky modules can really lead to leaks.

```
public class UnitTest2 {
1       @Before
2       public void setUp() throws Exception {
3       }
4       @Test
5       public void testBar(){
6           Test2 test = new Test2 ();
7           long memoryBefore = MemoryTester.getUsedMemory();
8           for(int i=0;i<MemoryTester.LARGE_LOOPNUM;i++){
9               test.bar();
10          }
11          long memoryAfter = MemoryTester.getUsedMemory();
12          MemoryTester.assertMemoryGrowth(memoryBefore,
memoryAfter, NO_SIGNIFICANT_GROWTH);
13      }
}
```

FIGURE 5 Unit testing with repeated method calls and memory assertions

Figure 5 shows a unit test for the example in Figure 2. We use the existing tools to generate the normal unit test. In the normal unit test, there only have the creation of Test2 object and a call to method bar. We generate the test code on the base of the normal unit test. Statement 8 puts method bar into a loop and sets a large number for the loop to strengthen memory use. Then, the memory growth assertions checks whether the memory growth is normal. If the growth is abnormal, it indicates the leak risky method may be leaking the memory.

5.3 CREATING UNIT TESTS FOR MODULES ABSORBING ARGUMENT OBJECTS

We use a weak reference based mechanism to determine whether the third kind of leak risky modules can cause problems.

```
public class UnitTest3{
1   @Before
2   public void setUp() throws Exception {
  }
  @Test
public void testZar(){
   Test3 test = new Test3();
   Object obj = new Data();
   WeakReference<Object>  ref= MemoryTester.prepareArgu-
ment(obj);
   test.zar(obj);
   obj = null;
   MemoryTester.assertArgumentNotLeaked (ref);
}
}
```

FIGURE 6 Unit testing with weak reference based leak detection mechanism

Figure 6 shows a unit test generated for the example in Figure 3. The paper firstly generates a normal unit test only containing the instantiation of Test3, the creation of an argument object, and a call to its method zar. Then, it generates strengthened unit test based on weak reference mechanism. A weak reference will be garbage collected when its referee is disconnected from other references. By checking whether a weak reference is broken, we can know whether an object is hold by other references. In the unit test, statement 8 uses MemoryTester.prepareArgument (objs) to add a weak reference ref to method zar's argument obj. It sets reference obj to null in statement 10. After that, if the argument obj is not absorbed by method zar, then the weak reference should be broken after some round of garbage collections, since there is no other reference to the argument object. Statement 11 does some GC and check the referee of the weak reference ref to judge whether object obj is absorbed and there can be leak source. By the weak reference checking mechanism, the unit test can determine whether the leak risky method may cause leaks.

6 Case studies

To validate the proposed approach, we implemented our approach as an Eclipse plugin and conduct case studies on several memory leak bugs in JDK which are typical examples of the risky modules introduced in section 3.

6.1 MODULES DIRECTLY LEAKING OBJECTS

The modules directly leaking objects are a little hard to find in the JDK memory leak bugs. But many existing bugs can easily be turn into this kind. For example, in Figure 7, we can easily get a representative directly leaking case based on a real memory leak bug JDK-6942989. In the case, there is a loop calling leaking method getAnonymousLogger() (the body of getAnonymousLogger() can be found in Figure 9). The execution rounds of the loop are unbounded and determined by the method's input. In method getAnonymousLogger(), the newly created Logger objects are added into an external container. In other words, the new objects escape from the loop scope. Therefore, the case matches the characteristics of the first kind of leak risky modules.

```
public class Worker {
 public static void doLoggedOperation(int n){
  for(int i = 0; i< n; i++){
       Logger logger = getAnonymousLogger();
       logger.log(record);
       doSth();
  }
}
}
```

FIGURE 7 A case for modules directly leaking objects

```
public class LoggerTest {
@Before
public void setUp() throws Exception {
}
@Test
public void testDoLoggedOperation () throws Exception{
Worker fixture = new Worker();
int n = MemoryTester.LARGE_INT;
long memoryBefore = MemoryTester.getUsedMemory();
fixture. doLoggedOperation(n);
long memoryAfter = MemoryTester.getUsedMemory();
          MemoryTester.assertMemoryGrowth(memoryBefore,
memoryAfter, NO_SIGNIFICANT_GROWTH);
}
}
```

FIGURE 8 Unit test for the case shows in Figure 7

Our approach firstly finds out the loop in method doLoggedOperation. Then, it identifies the loop is input dependent on method parameters. Finally, the approach determines that the newly created objects escape from loop scope. Therefore, the approach identifies the leak risky method doLoggedOperation and then generates the unit test. The unit test is shown in Figure 8. In the unit test, a large input data MemoryTester.LARGE_INT is set for tested method doLoggedOperation to strengthen memory use. When running the unit test, the results show that method doLoggedOperation consumes abnormal amount of memory and hence causes memory leaks. The case indicates our approach is effective in revealing the first kind of leak risky modules.

6.2 MODULES ACCUMULATING NEWLY CREATED OBJECTS

For this kind of leak risky modules, we use memory leak bug JDK-6942989: Memory leak of java.lang.ref.WeakReference objects as the studied case. The bug affects JDK version 4.2u27, 5.0u25, and 6. Its relevant code is briefly shown in Figure 9. In the case, a new WeakReference object created in the method doSetParent (indirectly called by method getAnonymousLogger()) is added into external container kids. Although the weak references are finally broken, there are still references from the external container to the new WeakReference objects. These objects in kids are not released on time. Therefore, the case matches the characteristics of the second kind of leak risky modules, and method getAnonymousLogger() has risk in leaking memory.

```
public class Logger {
   private boolean anonymous;
   private static Object treeLock = new Object();
   private Logger parent;
private ArrayList<LogManager.LoggerWeakRef> kids;
public static Logger getAnonymousLogger() {
  return getAnonymousLogger(null);
}
public static synchronized Logger getAnonymousLogger(String re-
sourceBundleName) {
……;
Logger result = new Logger(null, resourceBundleName);
result.anonymous = true;
Logger root = manager.getLogger("");
result.doSetParent(root);
return result;
}
private void doSetParent(Logger newParent) {
    ……;
    if (parent != null) {
            for (Iterator iter = parent.kids.iterator(); iter.hasNext(); ) {
              WeakReference ref = (WeakReference) iter.next();
Logger kid = (Logger) ref.get();
              if (kid == this) {
                  iter.remove();
                  break;
}
            }
 }
parent = newParent;
……;
parent.kids.add(new WeakReference(this));
   updateEffectiveLevel();
}
}
```

FIGURE 9 Code relevant to bug JDK-6942989

In our approach, we firstly do escape analysis with the escape sources in method getAnonymousLogger() excluding the return value and the parameters by Algorithm 2. Then, we can obtain all the escape objects. Finally, it identifies that the escape objects contain the newly created objects. Therefore, the approach considers method getAnonymousLogger() as risky. The unit test generated by our approach is shown in Figure 10. In the test, we put method getAnonymousLogger() into a loop and set a large upper bound for the loop to strengthen memory use. When running the unit test, the results show that the method getAnonymousLog-ger() can lead to memory leaks and indicate our approach can identify this kind of risky modules.

```
public class LoggerTest {
@Before
public void setUp() throws Exception {
}
@Test
public void testGetAnonymousLogger throws Exception (){
        long memoryBefore = MemoryTester.getUsedMemory();
        for(int i=0;i<MemoryTester.LARGE_LOOPNUM;i++)
Logger.getAnonymousLogger();
        long memoryAfter = MemoryTester.getUsedMemory();
        MemoryTester.assertMemoryGrowth(memoryBefore,
memoryAfter, NO_SIGNIFICANT_GROWTH);
    }
}
```

FIGURE 10 Unit test for case JDK-6942989

6.3 MODULES ABSORBING ARGUMENT OBJECTS

We use memory leak bug JDK-6525563: Memory leak in ObjectOutputStream as the studied case for the third category of risky modules. The bug affects JDK version 6. Its relevant code is briefly shown in Figure 11. In the case, a reference argument obj is passed to method writeObject0. The parameter obj is then added into an external array field objs of the receiver object in method insert which is indirectly called by method writeUnshared. Finally, the argument object will be absorbed inside method writeUnshared. The case matches the characteristics of the third kind of leak risky modules.

```
public class ObjectOutputStream{
private int size;
private int threshold;
private int[] spine;
private int[] next;
private Object[] objs;
public void writeUnshared(Object obj) throws IOException {
writeObject0(obj, true);
}
private void writeObject0(Object obj, boolean unshared) {
int h;
Object orig = obj;
Class cl = obj.getClass();
ObjectStreamClass desc = null;
writeOrdinaryObject(obj, desc, unshared);
}
private void writeOrdinaryObject(Object obj,  ObjectStreamClass
desc,  boolean unshared)
{
        assign(unshared ? null : obj);
}
int assign(Object obj) {
        insert(obj, size);
        return size++;
    }
private void insert(Object obj, int handle) {
        int index = hash(obj) % spine.length;
        objs[handle] = obj;
        next[handle] = spine[index];
        spine[index] = handle;
    }
private int hash(Object obj) {
        return System.identityHashCode(obj) & 0x7FFFFFFF;
    }
}
```

FIGURE 11 Code relevant to bug JDK-6525563

Our approach firstly marks the parameter object passed to method writeUnshared(Object obj) as a special new object by Algorithm 3. Then, we obtain all the escape objects by escape analysis. Finally, it can be found that the escape objects contain the special new object. In another words, the parameter object escapes from the method scope. Therefore, the approach recognizes method writeUnshared(Object obj) as a leak risky module and then gener-ates the final unit test for it (see Figure 12). In the unit test, it uses MemoryTester.prepareArgument(obj) to add a weak reference to the method's reference parameter and use MemoryTester.assertArgumentNotLeaked(ref) to test whether the weak reference have been broken to check memory leaks. While running the unit test, the results show that the test method writeUnshared(Object obj) causes

memory leaks. The case indicates our approach is able to test memory leaks for the third category of risky modules.

```
public class ObjectOutputStreamTest {
@Before
public void setUp() throws Exception {
}
@Test(expected = java.io.NotSerializableException.class)
public void testwriteUnshared() throws Exception{
ObjectOutputStream fixture = new ObjectOutputStream(new ByteAr-
rayOutputStream());
Object obj = new Data();
WeakReference<Object> ref = MemoryTester.prepareArgument(obj);
fixture.writeUnshared(obj);
obj = null;
    MemoryTester.assertArgumentNotLeaked (ref);
}
}
```

FIGURE 12 Unit test for JDK-6525563

The above cases show that the approach can identify and test three kinds of leak risky modules effectively. It can reveal real leaking methods, which could be helpful for practical use.

7 Conclusions

The paper proposes an approach for discovering memory leak phenomenon from testing perspective. The approach automatically generates unit tests to find potential memory leaks in Java methods. It firstly identifies three kinds of leak risky modules. Then, leak-oriented unit tests are generated from normal unit tests to strengthen the ability in finding leaks. We conduct case studies on real bugs. The results show that the approach is effective in revealing real leak modules. The paper focuses on unit testing. In the future, we also plan to do leak testing at system level to further support the discovering of memory leak bugs.

Reference

[1] Park J, Choi B 2012 Automated Memory Leakage Detection in Android Based Systems *International Journal of Control & Automation* **5**(2) 35-42

[2] Pienaar J A, Hundt R 2013 JSWhiz: Static analysis for JavaScript memory leaks *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*

[3] Jump M, McKinley K S 2010 Detecting memory leaks in managed languages with Cork *Software: Practice and Experience* **40**(1) 1-22

[4] Maxwell E K, Back G, Ramakrishnan N 2010 Diagnosing memory leaks using graph mining on heap dumps *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)* 115-24

[5] Aftandilian E E, Guyer S Z 2009 GC assertions: using the garbage collector to check heap properties *Proceedings of the 2009 ACM SIGPLAN conference on programming language design and implementation (PLDI)* 235-44

[6] Xu G, Bond M D, Qin F, Rountev A 2011 LeakChaser: helping programmers narrow down causes of memory leaks *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)* 270-82

[7] Yan D, Xu G, Yang S, Rountev A 2014 LeakChecker: Practical Static Memory Leak Detection for Managed Languages *International Symposium on Code Generation and Optimization (CGO)*

[8] Qian J, Zhou X 2012 Inferring Weak References for Fixing Java Memory Leaks *The 28th IEEE International Conference on Software Maintenance (ICSM)* ERA Track 571-4

[9] Kim D, Nam J, Song J, Kim S 2013 Automatic patch generation learned from human-written patches *The 2013 International Conference on Software Engineering (ICSE)*

[10] Zhang S, Lü H, Ernst M D 2013 Automatically repairing broken workflows for evolving GUI applications *International Symposium on Software Testing and Analysis (ISSTA)*

[11] JProbe. http://www.quest.com/jprobe/

[12] O'Hair K 2004 HPROF: A Heap / CPU Profiling Tool in J2SE 5.0. http://java.sun.com/developer/technicalArticles/Programming/HPROF.html

[13] CodeProAnalytix. https://developers.google.com/java-dev-tools/codepro/doc/

[14] Xu B, Qian J, Zhang X, Wu Z, Chen L 2005 A brief survey of program slicing *ACM SIGSOFT Software Engineering Notes* **30**(2) 10-45

[15] Hind M 2001 Pointer analysis: Haven't we solved this problem yet? *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*

[16] Gay D, Steensgaard B 2000 Fast escape analysis and stack allocation for object-based programs *Proceedings of the 9th International Conference on Compiler Construction* 82-93.

[17] Vallée-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V 1999 Soot – a Java bytecode optimization framework *Proceeding CASCON '99 Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research* 13-23

Authors

Lijuan Hong, born in March, 1989, Hefei, China

University studies: MS degree in Computer Science and Technology at Nanjing University of Aeronautics and Astronautics, China, 2012.
Scientific interests: program analysis, program tests.

Ju Qian, born in 1981, Nanjing, China

Current position, grades: associate professor at Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu, China.
University studies: PhD degrees in Computer Science and Technology at Southeast University, China.
Scientific interests: program analysis, program tests, program diagnostics.
Publications: 4 papers.

Jifeng Cui, born on March 16, 1988, Nanjing, China

University studies: MS degree in Computer Science and Technology at Nanjing University of Aeronautics and Astronautics, China, 2011.
Scientific interests: data mining, big data analysis.
Publications: 3 papers.