Qian J, Zhou D. Prioritizing test cases for memory leaks in Android applications. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 31(5): 869–882 Sept. 2016. DOI 10.1007/s11390-016-1670-2

Prioritizing Test Cases for Memory Leaks in Android Applications

Ju Qian[1,2,3], *Member, CCF*, and Di Zhou[1,3]

[1] *College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China*
[2] *State Key Laboratory of Software Engineering, Wuhan University, Wuhan 430072, China*
[3] *Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing 210023, China*

E-mail: jqian@nuaa.edu.cn; dzhou0216@163.com

Received March 20, 2016; revised July 17, 2016.

Abstract Mobile applications usually can only access limited amount of memory. Improper use of the memory can cause memory leaks, which may lead to performance slowdowns or even cause applications to be unexpectedly killed. Although a large body of research has been devoted into the memory leak diagnosing techniques after leaks have been discovered, it is still challenging to find out the memory leak phenomena at first. Testing is the most widely used technique for failure discovery. However, traditional testing techniques are not directed for the discovery of memory leaks. They may spend lots of time on testing unlikely leaking executions and therefore can be inefficient. To address the problem, we propose a novel approach to prioritize test cases according to their likelihood to cause memory leaks in a given test suite. It firstly builds a prediction model to determine whether each test can potentially lead to memory leaks based on machine learning on selected code features. Then, for each input test case, we partly run it to get its code features and predict its likelihood to cause leaks. The most suspicious test cases will be suggested to run at first in order to reveal memory leak faults as soon as possible. Experimental evaluation on several Android applications shows that our approach is effective.

Keywords Android, memory leak, test case prioritization, test execution

1 Introduction

With the growing use of mobile devices, the Android system and the mobile applications running on it have become increasingly popular. The wide use of mobile applications imposes great demands on mobile software quality. Unfortunately, many Android applications still suffer from various bugs. Memory leak is a typical performance-related bug. Even with garbage collection supports, memory leak still remains a problem for Android programs. Since Android programs running on mobile devices usually can only access limited amount of memory resources, the problem could be serious. Those memory leaks can lead to performance slowdowns or cause applications to be unexpectedly killed by the OS due to resource pressure, which affects user experience. By searching in Stack Overflow①, we can find that there are over 2 300 questions about keyword combination of "Android" and "memory leak", compared with about 4 100 questions found by the combination of "Java" and "memory leak". This indicates that effective approaches to helping resolve memory leaks in Android applications are needed.

To solve the memory leak problem in garbage-collected languages, a variety of techniques and tools have been proposed[1-15]. Some use static approaches[2-3], while most of the work prefers dynamic ones[5-15] since dynamic approaches tend to be more precise. Of the dynamic techniques, a large body of work focuses on memory leak diagnosing[4-11] after obtaining some suspicious test executions, while very little

Regular Paper

Special Section on Software Systems 2016

This work was supported by the Defense Industrial Technology Development Program of China under Grant Nos. JCKY2016-206B001 and JCKY2014206C002, the Open Project of State Key Laboratory of Software Engineering of Wuhan University under Grant No. SKLSE2014-10-13, and the Fundamental Research Funds of Nanjing University of Aeronautics and Astronautics under Grant No. NS2013088.

① Stack Overflow. http://stackoverflow.com, July 2016.

work focuses on discovering leak executions. Without discovering memory leak phenomena at first, even the best leak diagnosing techniques could not be applied.

Discovering potential memory leak phenomena in Android applications is still challenging[12-15]. Recently, Yan *et al.*[13-14] presented an approach to generate test cases to discover memory leaks and other resource leaks in Android applications based on neutral cycles in GUI model. Shahriar *et al.*[15] presented another approach that uses fuzzing techniques to test memory leaks in Android programs according to several leak patterns specific to Android platform. Although effective, these approaches may generate a large number of test cases for complex applications. Since each test case may take minutes to hours of execution time to trigger hundreds of repeated GUI events in order to clearly observe the memory behavior, the testing can be costly. The experimental study in [13] shows that only a few test cases may really lead to memory leaks. Therefore, testing all the generated test cases can be unnecessary. A tester may waste lots of time on testing unlikely leaking executions before discovering a leaking execution that can be used for diagnosing.

To address the problem, we propose a novel approach to prioritize test cases in a given test suite according to their likelihood to cause memory leaks. Each test case in the test suite is assumed to be composed by a number of looped GUI interaction sequences (usually only the long repeating sequence can cause noticeable memory leaks). The approach firstly uses a set of training test cases to build a memory leak prediction model, which is used to determine whether a test case can potentially lead to memory leaks by using machine learning on leak-relevant code features. The code features are extracted according to the occurrences of Java language memory management instructions, the use of system resources, and the use of application framework resources in a program. Then, for each test case in the given test suite, we partially run one round of the GUI interaction loop to collect its leak-relevant code features, and predict its likelihood to cause leaks with the leak prediction model. All the test cases will be prioritized according to the likelihood score and reported to the user. A user can run those most suspicious test cases at first so that the memory leak faults can be revealed as soon as possible. We apply the proposed approach on a set of real-world open-source Android applications. The initial results show that the test cases containing memory leaks can be ranked in high order. This suggests that our approach can be valuable.

The rest of the paper is organized as follows. Section 2 introduces the technique background; Section 3 and Section 4 present an overview of the whole approach and the memory leak relevant code features respectively. We propose a memory leak prediction model and describe its usages in Section 5. Section 6 is the experimental study. Finally, we discuss the related work and conclude the paper in Section 7 and Section 8 respectively.

2 Background

2.1 Structure of Android Applications

An Android application is composed of several types of Java components instantiated at run-time, including the activities, services, broadcast receivers, content providers, etc. The activities manage the hierarchies of GUI widgets, e.g., View, ViewGroup, Widget, Menu, and Dialog. They are the key part for the user interactions in an application.

An application can have one or more activities. The application's lifecycle is mostly determined by the lifecycle of the activities. As demonstrated in Fig.1, each activity has a well-defined lifecycle passing through three main states: running, paused and stopped. Developers can define callback methods to handle these three states. Once an activity is launched, the `onCreate()` method will be called, while when the activity is ready to terminate, the `onDestroy()` method will be called. The loop between `onStart()` and `onStop()` is a lifetime that the activity is visible to users. An activity can call other activities dynamically, and this will cause the calling activity to pass to the paused state.

2.2 Memory Leaks in Android Applications

Each Android application is independently running in a separate OS process, inside an instance of a Dalvik VM, with a memory limit. Like Java programs, Android applications mainly create objects in heap space. If an object becomes unreachable from the heap roots, the object will be freed and its memory will be reclaimed by a garbage collector. However, when an application maintains a reference to an object, even if the object is no longer used, the garbage collector could not reclaim its memory. This can cause memory leaks easily.

The improper allocation and deallocation of space in native memory and the improper use of other resources like threads, bitmaps, may also cause the unexpected

increase of the used memory. In this paper, we consider all problems that can lead to the unreclaimable memory in a program as memory leak problems.

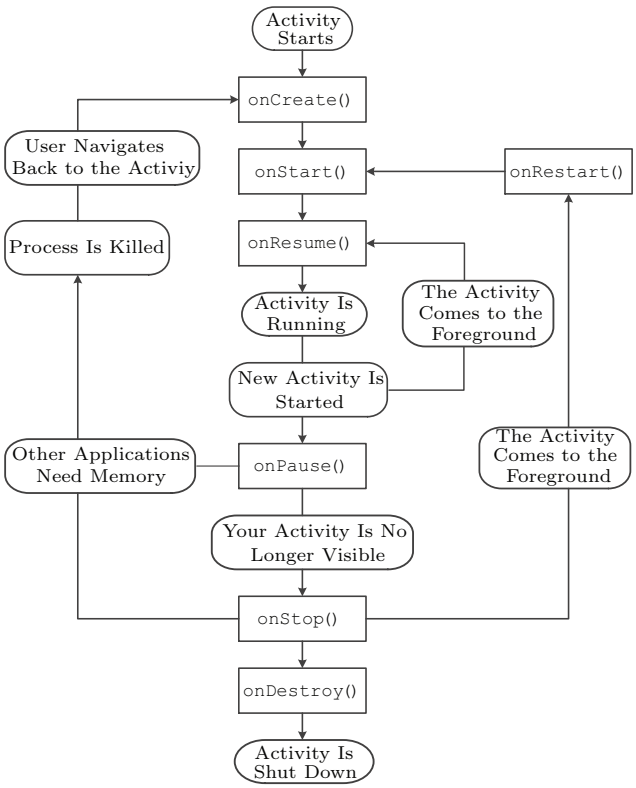


Fig.1. Activity lifecycle[12].

Because Android programs run on a different virtual machine and a different application framework (Android SDK) and have a different lifecycle compared with normal Java programs, the resolution of memory leaks in Android programs will be different from that in traditional Java programs.

2.3 Discovering Memory Leaks in Android

Yan *et al.*[13] proposed an automated test cases generation approach to discover memory leaks in Android applications. The approach is based on neutral cycles in an application's GUI model. A neutral cycle is a sequence of GUI events expected to have a "neutral" effect, and does not lead to the increases in resource usage. For example, in a PDF reader, there can be zoom-in/out operations which can enlarge or shrink the PDF display screen. A sequence of GUI events composed by these two zoom operations should be a neutral cycle. Yan *et al.*[13] defined coverage criteria based on neutral cycles, and proposed methods to generate GUI operation sequences with neutral cycles as test cases. By discovering previously unknown memory leak bugs, the approach has been proven effective on many applications.

However, a comprehensive testing approach based on neutral cycles may produce a large test suite for a complex application. Assume there are N_{testcase} test cases in the test suite, and each test case runs N_{leading} GUI operations before entering a neutral cycle and N_{loop} neutral cycles, each consisting of $N_{\text{cycle\_op}}$ GUI operations. Then, to execute the whole test suite, we may need to execute

$$N_{\text{operation}} = N_{\text{testcase}} \times (N_{\text{leading}} + N_{\text{loop}} \times N_{\text{cycle\_op}})$$

GUI operations. That can be very costly even if these operations are triggered by a test automation framework like Robotium②.

We are aware that often there are only a few test cases which really lead to memory leaks. Without carefully scheduling the execution orders, large testing efforts may be wasted on those test cases unlikely to cause leaks. If the test cases potentially causing leaks are prioritized and tested before those unlikely leaking ones, then the leak discovering efficiency can be effectively improved.

To address the above problem, we propose a test case prioritization approach to discover memory leaks, which will be elaborated in the following sections.

3 Test Case Prioritization Framework

An overview of our test case prioritization framework is shown in Fig.2. The framework firstly accepts a set of training inputs to build a memory prediction model. Then we input an application under test as well as a test suite for the application to the framework. A prioritized sequence of test cases will be outputted to the users for memory leak testing.

3.1 Training Inputs

The training inputs include a set of applications used for training, a collection of test cases toward these applications, and a leak level function that records the leak level of the training test cases.

A training test case is composed by a sequence of GUI events like that in [13], which can be expressed in

②Robotium. http://www.robotium.org, July 2016.

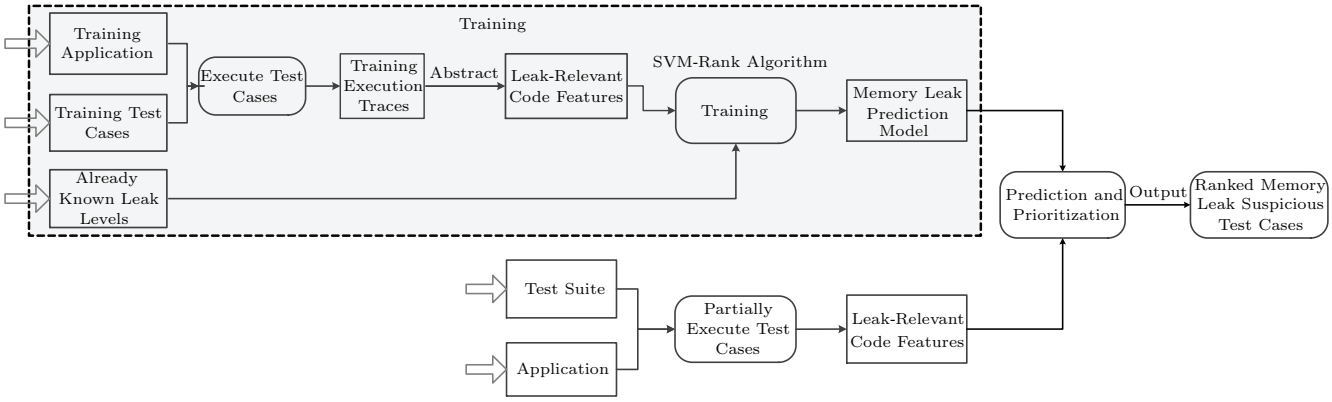


Fig.2. Test case prioritization framework.

a form like

$$(n_0, \ldots, n_i, (n_j, \ldots, n_i)^k),$$

where n_0 is the start event, the prefix (n_0, ..., n_i) represents a cycle-free leading event sequence, and (n_j, ..., n_i)^k is event cycles returning to n_i each time. Fig.3 demonstrates the structure of a test case. In the figure, n_0 represents a GUI operation in the application's start screen. (n_0, ..., n_i) represents a sequence of GUI operations before entering event cycles. (n_j,..., n_i)^k is the event cycles conducting k rounds of repeated GUI actions. The training test cases will be fully executed to collect training code features that are relevant to memory leaks.

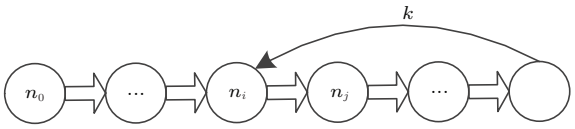


Fig.3. Demonstration of a test case.

Each training test case is labeled with a leak level to indicate whether there are memory leaks in the test case and if there is no memory leak then how close the non-leaking test case is to the leaking ones. The labelling of leak levels can be expressed in the following function:

$$LL : T_{\text{train}} \rightarrow \{1, ..., K\}.$$

A test case with known memory leaks will be labeled with the maximum value K in the function range. For a non-leaking test case, we get the grown memory size after repeating a number of event cycles in the test case. The ones with almost no memory growth will be labeled with 1. Others will be labelled with a value between 1 and K according to a series of bars set for the grown memory size. A test case with larger grown memory size is considered to be closer to the leaking test cases.

3.2 Test Suite for Prioritization

The test cases for prioritization are similar to those in the training set. These cases will be scored by a leak prediction model. According to scores, they will be prioritized in an order from high score to low score.

In our approach, we need to partially run each test case in the given test suite to collect memory leak relevant code features for prioritization. However, different from running test cases in the training set, we only need to execute the event cycle (n_j, ..., n_i) in a test case for prioritization for only one cycle. Although this cannot completely avoid running some test cases in the test suite, it can still reduce significantly the cost of testing.

Similar to Subsection 2.3, assume there are N_{testcase} test cases in the test suite for prioritization, and each test case runs N_{leading} GUI operations before entering an event cycle and N_{loop} event cycles, each consisting of $N_{\text{cycle\_op}}$ GUI operations. Then, to do prioritization, we need to run

$$N_{\text{prioritization}} = N_{\text{testcase}} \times (N_{\text{leading}} + N_{\text{cycle\_op}})$$

additional GUI operations before fully executing each test case. Let $N_{\text{testcase}} = 100$, $N_{\text{leading}} = 4$, $N_{\text{loop}} = 30$, $N_{\text{cycle\_op}} = 3$, and suppose before prioritization we can find a leak execution and stop testing in 1/2 of total test cases and after prioritization we can find a leak execution and stop testing in 1/5 of total test cases. Then, before prioritization we need to run 4 700 GUI operations in testing, while after prioritization the number can be reduced to 2 580. About 45% of GUI operations can be reduced by doing prioritization. If the

prioritization precision is improved, more improvement can be achieved. This yet has not considered the time that a user takes to check the results of each test case.

3.3 Prioritization Steps

Our prioritization includes two major steps: training and ranking. In the training step, we build a simple memory leak prediction model based on memory leak relevant code features extracted from an input test case. The model can produce a score to indicate the likelihood that a test case may cause memory leaks. It has a weight vector to determine the importance of each leak relevant feature. We use machine learning on the features extracted from the training test cases and the leak levels of those test cases to obtain the weight vector. With the code features and the weight vector, the test cases in a test suite can be prioritized.

In this paper, we use the code-level features for memory leak prediction. Other choices might be using the memory growth in one GUI operation cycle or the leak reports produced by a leak detection tool like MAT[③] for one GUI operation cycle as leak-relevant execution features. However, investigations on several applications show that such choices are almost infeasible under our prioritization framework. The memory growths in one GUI operation cycle are usually very small and easy to be affected by the environment. Therefore, it is very difficult to build a precise prediction model based on such information. Using the leak reports from a tool like MAT has similar problems. The prediction performance of such leak detection tools is often unstable when the used memory dump is very small. The work in [16] also finds that MAT suffers from false alarms for many Android applications. Different from these choices, code-level features do not have such limitations. They can predict whether there are memory leaks even if the amount of leaked memory is very small.

The ranking step uses the memory leak prediction model to prioritize test cases. Each test case in the given test suite is partially run to collect a leak-relevant code feature vector, the leak prediction model gives scores with the code feature vectors, and then the test cases are sorted according to the scores.

4 Memory Leak Relevant Code Features

We identify three categories of features that are relevant to memory leaks in a program: the use of Java language memory management instructions, the use of Java runtime system resources, and the use of application framework resources. While there are many different features in each category, we simply choose a part of frequently used features. The used code features are listed in Table 1. We will introduce them in more details in this section. Based on the memory leak relevant code features, we build feature vectors and then use them to prioritize test cases.

Table 1. Categories of the Memory Leak Relevant Code Features

	Category	Code Feature
1	Java language memory management instructions	`new/newarray`
		nullify
2	Use of Java runtime system resources	`openXXX()`
		`closeXXX()`
3	Use of application framework resources	`getResources()`
		`onDestory()`

4.1 Use of Java Language Memory Management Instructions

The Java language memory management instructions refer to the `new/newarray` instructions and the reference target resetting instructions. When a `new` or `newarray` instruction is executed, more memory is allocated on the heap. However, some memory can possibly be freed if a reference resetting instruction, especially a nullifying instruction, is executed. These instructions can be indicators for potential memory leaks. If a program executes too many `new/newarray` instructions, then the likelihood of containing memory leaks can possibly be higher. If there are more reference nullifying instructions executed in a program, we consider that the programmer has handled memory management more carefully, and hence the likelihood of containing memory leaks can be lower.

For example, in Fig.4, there is a `newarray` instruction corresponding to the `new` statement. After being assigned to instance field `data`, the created array might be referenced by some long living reference and cannot be garbage-collected. Such code snippet is considered more likely to contain a memory leak bug, compared with a module without creating any object via `new/newarray` instructions.

We can use the number of occurred `new/newarray` instructions and the number of nullifying instructions as a heuristic for predicting memory leaks. The Java

③Memory Analyzer (MAT). http://www.eclipse.org/mat/, July 2016.

language level code features are expressed in the following vector:

$$\boldsymbol{F}_{\text{language}}(t) = (N_{\text{new}}, N_{\text{newarray}}, N_{\text{null}}),$$

where N_{new} is the number of `new` instructions, N_{newarray} is the number of `newarray` instructions, and N_{null} is the number of nullifying instructions.

```
public class Test1 extends Activity {
  ...
  public void onCreate(Bundle bundle){
    super.onCreate(bundle);
    setContentView(R.layout.Test1);
    ...
    data = new int[1024];
    ...
  }
}
```

Fig.4. Example module with `newarray` instruction.

4.2 Use of Java Runtime System Resources

There can also be memory leaks in an Android application if the Java runtime system resources, e.g., files or databases, are not released properly, even though the leaked memory may occur in the native side of the system. For example, in Fig.5, after opening a file, if there are some errors, the file close action may not be executed, which may cause the file to long termly occupy memory and eventually lead to a memory leak.

The open and close actions can be a heuristic for predicting potential memory leaks. If the execution trace of an application contains many open actions but few close actions, then the execution is considered more likely to contain a memory leak.

Therefore, the use of Java runtime system resources is also considered as part of the memory leak relevant code features. We simply count the `openXXX()` like method calls and the `closeXXX()` like method calls in the execution trace. The counted numbers form a Java runtime resource relevant code feature vector, which can be expressed in the following formula:

$$\boldsymbol{F}_{\text{system}}(t) = (N_{\text{open}}, N_{\text{close}}),$$

where N_{open} is the number of open calls, while N_{close} is the number of close calls. All the counts of different `openXXX()` like calls are combined as a single feature, and the same for `closeXXX()` like calls. The feature vector consisting of open/close call numbers may not comprehensively represent all the use of Java runtime system resources. It can be further extended when applied in more complex systems.

```
public class Test2 extends Activity{
  public void onCreate(Bundle bundle){
   super.onCreate(bundle);
   setContentView(R.layout.Test2);
     ...
   try{
       fis = open FileInput(fileName);
       ...
       fis.close();
   }
  }
}
```

Fig.5. Example module opening and closing a file.

4.3 Use of Application Framework Resources

In addition to allocating memory via `new/newarray` instructions and via Java runtime resource management APIs, memory can also be indirectly allocated by the application framework resource management APIs, e.g., when loading Bitmap or ImageView objects in the Android framework.

For example, in Fig.6, a Bitmap image used as the background of a GUI screen is loaded via a `BitmapFactory.decodeResource(getResources(), R.drawable.bimap)` call and saved into a `Bitmap` object. The bitmap maintains a large amount of memory. It should be carefully released; otherwise, there can possibly be memory leaks.

Similar to Subsection 4.1 and Subsection 4.2, we can count the numbers of application framework resource allocation and release operations as code features to help predict memory leaks. However, there are too many kinds of resources in the Android framework, and their allocation and release operations are quite different. Directly counting the numbers of such operations could be very complex. We are aware that many of the Android framework resource allocations directly or indirectly call a `getResources()` method provided by the Android API to load application resources. Therefore,

we use the number of `getResources()` method calls in an execution trace as the code feature indicating resource allocations. If the number of `getResources()` calls is large, we choose to treat this as a suspicious symptom that may be related to a memory leak. Besides, we are also aware that many resources are often released in the `onDestroy()` methods. To simplify code feature collection, we use the number of `onDestroy()` method calls as a code feature to approximate the occurrences of resource release operations.

```
public class Test3 extends Activity {
    Bitmap bitmap;
    public void onCreate(Bundle bundle){
        super.onCreate(bundle);
        setContentView(R.layout.Test3);

        // Access to bitmap resource
        bitmap = BitmapFactory.decodeResource
            (getResources(), R.drawable.bimap);
    }
    public void onDestroy(){
        ...
      super.onDestroy();
    }
}
```

Fig.6. Example module using application framework resources.

Let $N_{\text{getResources}}$ and $N_{\text{onDestroy}}$ be the numbers of `getResources()` and `onDestroy()` calls, respectively. Then, the memory leak relevant code features from the application framework resource perspective are shown as follows.

$$\boldsymbol{F}_{\text{framework}}(t) = (N_{\text{getResources}}, N_{\text{onDestroy}}).$$

4.4 Extracting of Code Feature Vector

To extract memory leak relevant code features, we firstly instrument the application under test to collect the execution trace of each test case. Then, the code features will be extracted from the execution trace. The process is shown in Fig.7.

In the instrumentation step, we instrument the Android applications at bytecode-level with Soot bytecode analysis framework[17]. Hooks are inserted before the instructions that are related to the code features to collect the executed instructions. After running the given test case on the instrumented application, an execution trace with information relevant to the identified code features will be obtained. The code features can be extracted by simply scanning the execution trace. During the scanning, only the part of an execution trace that corresponds to the event cycles in a test case will be considered, since usually only these event cycles are responsible for the memory leaks in a test execution.

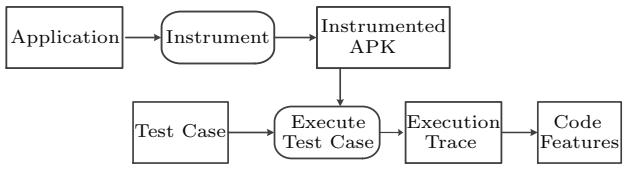


Fig.7. Extracting code features.

We currently do not spend many efforts in optimizing the execution profiling. When testing in practice, the code features extraction cost can be further reduced. The designed code features are not very comprehensive. However, they cover main categories of code-level features that are relevant to memory leaks in an Android application. One may further add features to each category if necessary to extend its capability of predicting memory leaks.

5 Leak Prediction Model

We use a linear combination of all code feature values to predict the likelihood of leaks. In the prediction model, the weight values of the linear combination are determined by a machine learning algorithm.

5.1 Building a Leak Prediction Model

Let $T_{\text{prioritize}} = \{t_1, t_2, \ldots, t_p\}$ be a test suite of an application P for prioritization. After executing an event cycle in each test case $t_i \in T$, we can obtain a feature vector $\boldsymbol{F}_{\text{memory}}$ for t_i to describe its leak relevant code-level execution characteristics. The vector can be denoted as below, in which m is the length of a test case's full feature vector.

$$\begin{aligned}\boldsymbol{F}_{\text{memory}}(t_i) &= (\boldsymbol{F}_{\text{language}}(t_i), \boldsymbol{F}_{\text{system}}(t_i), \boldsymbol{F}_{\text{framework}}(t_i)) \\ &= (f_1, f_2, ..., f_m).\end{aligned}$$

Our leak prediction model is based on a ranking function which can be represented as a linear combination of the feature values. The ranking function S is presented in the following formula. It is a mapping

from a test case to its leak likelihood score. We multiply a feature vector with a weight vector $\boldsymbol{W}$ to get the score value.

$$S(t) = \boldsymbol{F}_{\text{memory}}(t) \cdot \boldsymbol{W} = (f_1, f_2, ..., f_m) \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{pmatrix}.$$

In prioritization, the leak likelihood scores are calculated for each test case in test suite $T_{\text{prioritize}}$. According to the scores, the test cases will be sorted in descending order as the prioritization result. Test cases in the front of the result list are the ones likely to cause memory leaks and are suggested to be run at first. The scoring and prioritization process is depicted in Fig.8.

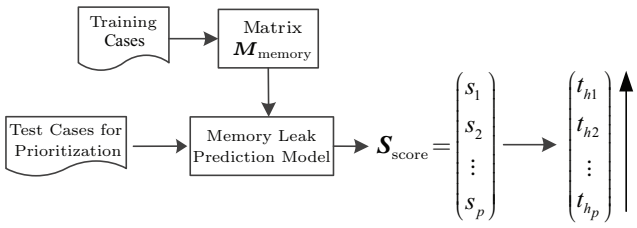


Fig.8. Process of the test case prioritization.

5.2 Determining the Feature Weights

Now the remaining problem is to determine the feature weights $\boldsymbol{W}$. We obtain reasonable feature weights by machine learning. Among many machine learning algorithms and models, the `Ranking SVM` algorithm[18] is found suitable for our memory leak prediction model. Ranking SVM is a variant of the support vector machine algorithm, which is used to solve certain learning to rank problems. It outputs a weight vector which can be multiplied with the feature vector as a linear ranking function. The approach is originally used for information retrieval, but can also be used in other areas.

To use the Ranking SVM algorithm, we firstly extract the code features for each test case in the training set. Let there be n training test cases for all the training applications, and then the training test set can be represented by:

$$\boldsymbol{T}_{\text{train}} = (t_1, t_2, ..., t_n).$$

After executing all the event cycles in a training test case t_k, we can obtain a code feature vector $\boldsymbol{F}_{\text{memory}}(t_k)$ for t_k. According to priori knowledge, each training test case t_k will be assigned with a leak level $LL(t_k)$. Based on such information, a memory leak relevant feature-level matrix $\boldsymbol{M}_{\text{memory}}$, which contains all the feature vectors and the leak levels, can be built, as shown in Fig.9. In $\boldsymbol{M}_{\text{memory}}$, each line records the feature vector and the memory leak level of a training test case.

$$\boldsymbol{M}_{\text{memory}}(P) = \begin{pmatrix} \boldsymbol{F}_{\text{memory}}(t_1), LL(t_1) \\ \boldsymbol{F}_{\text{memory}}(t_2), LL(t_2) \\ \vdots \\ \boldsymbol{F}_{\text{memory}}(t_n), LL(t_n) \end{pmatrix} = \begin{pmatrix} f_{11}, f_{12}, ..., f_{1m}, l_1 \\ f_{21}, f_{22}, ..., f_{2m}, l_2 \\ \vdots \\ f_{n1}, f_{n2}, ..., f_{nm}, l_n \end{pmatrix}$$

Fig.9. Feature-level matrix.

$\boldsymbol{M}_{\text{memory}}$ forms a basis for the Ranking SVM algorithm. The algorithm then uses a pairwise approach to transform the feature-level information into the binary classification information. Fig.10 demonstrates the transformation. For every two training test cases t_a and t_b, if the leak level of t_a is higher than that of t_b, i.e., $LL(t_a) > LL(t_b)$, then the test case pair (t_a, t_b) will be classified into a group with label "+1". Otherwise, if $LL(t_a) < LL(t_b)$, the test case pair will be classified into a group with label "−1". The feature vector of test case pair (t_a, t_b) will be $(\boldsymbol{F}_{\text{memory}}(t_a), \boldsymbol{F}_{\text{memory}}(t_b))$. Then, the feature-level information of each test case becomes the feature-classification information of each test case pair.

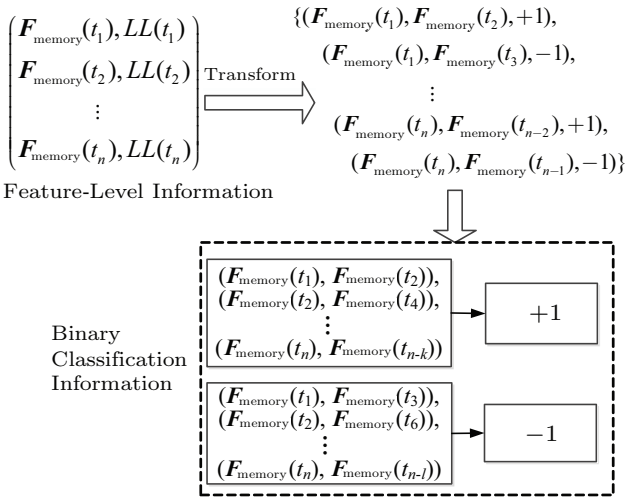
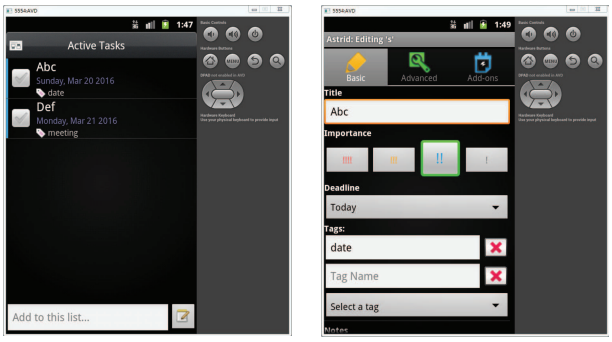


Fig.10. From sorting to binary classification problem.

The Ranking SVM algorithm will use classical SVM algorithms to learn a linear classification function that fits the given binary classification model. A ranking function consistent with the leak level information in the training set can be derived from the classification function. By transforming the ordered data (order by leak levels) into binary classification data, the Ranking SVM algorithm can turn the sorting problem into the binary classification problem. After obtaining the ranking function, we can get the weight vector suitable for our memory leak prediction.

5.3 Using the Prediction Model

We will use an example to show how the prediction model works for an Android application under test. The tested application is Astrid④, which implements a task management function. Fig.11 shows two activities in Astrid. Fig.11(a), TaskListActivity, displays a list of tasks when the application is launched. Fig.11(b) shows the task editing interface. Tapping the task icons on the TaskListActivity can switch to TaskEditActivity, while pressing the Android system BACK button can stop editing and return to the previous screen.



(a) (b)

Fig.11. Astrid application. (a) TaskListActivity: listing tasks. (b) TaskEditActivity: editing a task.

A GUI operation sequence consisting of tapping a task icon at first and then pressing the BACK button to go back forms an event cycle. There can be a test case containing this cycle. When executing the test case, we can collect a memory feature vector for this event cycle. The feature vector is:

$$(461, 65, 56, 1, 9, 31, 0).$$

In this vector, the numbers of `new` instructions, `newarray` instructions, nullifying instructions, `openXXX()` calls, `closeXXX()` calls, `getResources()` calls, and `onDestroy()` calls are 461, 65, 56, 1, 9, 31, and 0, respectively. By inputting this feature vector to the leak prediction model, we can get a leaking score 35.73 for the test case. The score is much higher than the scores of non-leaking test cases, which indicates the test case is more likely to cause memory leaks.

6 Experiments

We implemented our approach and evaluated it on several publicly available Android applications to validate whether the prioritization results can really be beneficial for the users.

6.1 Experimental Setup

In our experiments, all the applications were run on an Android virtual device. The test cases were executed based on Robotium test automation framework. Each application was instrumented with Soot 2.5.0 to obtain the execution traces. The leak relevant code feature vectors were then extracted from the recorded execution traces.

In the training step, we defined three leak levels {1, 2, 3} for the executed test cases. Level 3 corresponds to the test cases with known memory leaks, e.g., the leaking executions identified by the work in [13]. Level 1 corresponds to test cases with almost no memory increase after 10 cycles of operations (the size of the increased memory is smaller than 10 Kb). The other test cases were classified into level 2. We used DDMS tool⑤ to monitor the change of heap memory so that the leak levels can be determined.

After obtaining the code feature vectors and the leak level labels, we then built a model for memory leak prediction with the help of SVM-rank⑥, an implementation of the Ranking SVM algorithm. Each test case for prioritization was run for one GUI operation cycle to collect its leak relevant code feature vector. The vector would then be inputted to the memory leak prediction model to determine its likelihood to lead to memory leaks. The test cases in the test suite for prioritization would be sorted according to their leak likelihood score. Those test cases with high scores were suggested to be

④Astrid. https://github.com/todoroo/astrid, July 2016.

⑤DDMS. https://developer.android.com/studio/profile /ddms.html, July 2016.

⑥SVM-rank. http://www.cs.cornell.edu/People/tj/svm_light /svm_rank.html, July 2016.

tested before other ones so that memory leak phenomena can be discovered as soon as possible. We checked the ranked positions of the test cases known to have memory leaks to validate the effectiveness of the proposed approach.

The experiments were conducted on nine open-source Android applications. Most of them were obtained from the subjects in previous work[13] (downloaded from the authors' website⑦). Because we had failed to produce runnable instrumented APKs for applications ConnectBot, FBReader, and K9 used in [13] under Soot framework, to complement the benchmark applications, other four applications (Omnidroid 0.2.2⑧, Open Manager 1.11⑨, Sipdroid 2.2 beta⑩, SuperGenPass 2.2.3⑪) were also picked for study. These applications have been used in other software engineering researches[19-20]. For the subjects from [13], we used the same test cases for study. For the other subjects, we manually generated test cases following the neutral-cycle based test case generation approach. During testing, we found several generated test cases cause the subject applications' memory consumption to significantly increase after multiple neutral cycles. These test cases exhibited memory leak behaviors if too many neutral cycles were triggered. We considered them as leak revealing test cases that should be prioritized to the front for testing, compared with the test cases that only consume ignorable memory after neutral cycles.

The experimental subjects are listed in Table 2. The third column shows the number of activities in the applications. The fourth column shows the number of classes. The fifth column shows the number of test cases existing for each application. In most test cases, the numbers of events involved in the event cycles are less than 5. The sixth column shows the number of test cases that can reveal memory leaks (as stated in Subsection 2.2, the resource leaks finally causing abnormal memory growth are also considered as memory leaks in this paper).

Table 2. Experimental Subjects

Application	Description	# Activity	# Class	# Test Case	# Leak Test Case
APV	PDF readers	4	56	22	1
Astrid	To-do list management	11	481	40	3
KeePassDroid	Password manager	7	126	33	5
Omnidroid	Automated event/action manager	13	163	18	2
Open Manager	File management	8	60	26	1
Sipdroid	Network phone	12	331	38	1
SuperGenPass	Bookmarklet password generator	3	65	16	1
VLC	Multimedia player	8	176	32	4
VuDroid	ebook	3	67	17	2

We used cross-validation to validate the effectiveness of our approach. The experimental subjects were divided into three groups: {Astrid, Omnidroid, VLC}, {APV, KeePassDroid, VuDroid}, and {Open Manager, Sipdroid, SuperGenPass}. We did experiments three times according to these three groups. Each time a group of subjects were selected for training, while the other subjects were used for test case prioritization. The three experiments are listed as below.

Experiment 1. Training apps: Astrid, Omnidroid, VLC.

Experiment 2. Training apps: APV, KeePassDroid, VuDroid.

Experiment 3. Training apps: Open Manager, SipDroid, SuperGenPass.

6.2 Experimental Results

The indicators of the effectiveness of the proposed approach include: the best rank (the highest rank), the worst rank (the lowest rank), the median rank of the leaking test cases, the percentage of test cases with ranks greater than or equal to the best rank, the percentage of test cases with ranks greater than or equal to the worst rank, and the percentage of test cases with ranks greater than or equal to the average rank. The

⑦http://web.cse.ohio-state.edu/presto/software/leakdroid/, Aug. 2016.

⑧Omnidroid. https://f-droid.org/repository/browse/?fdid= edu.nyu.cs.omnidroid.app, July 2016.

⑨Open Manager. https://f-droid.org/repository/browse/?fdid= com.nexes.manager, July 2016.

⑩Sipdroid. https://f-droid.org/repository/browse/?fdid= org.sipdroid.sipua, July 2016.

⑪SuperGenPass. https://f-droid.org/repository/browse/?fdid=info.staticfree.SuperGenPass, July 2016.

rank data for the three experiments are listed in Table 3∼Table 5. Columns of Best, Worst, Med, Best (%), Worst (%), and Avg (%) show the above indicator values, respectively. The notes below these tables also show the weight vector in the memory leak prediction model of each experiment. The weight vectors are instances of code feature vector:

$$(N_{\text{new}}, N_{\text{newarray}}, N_{\text{null}}, N_{\text{open}}, N_{\text{close}}, N_{\text{getResources}}, N_{\text{onDestroy}}).$$

Table 3. Results of Experiment 1

Application	Best	Worst	Med	Best (%)	Worst (%)	Avg (%)
APV	6.0	6.0	6.0	27.3	27.3	27.3
KeePassDroid	1.0	8.0	4.0	3.0	24.2	12.7
Open Manager	2.0	2.0	2.0	7.7	7.7	7.7
Sipdroid	1.0	1.0	1.0	2.6	2.6	2.6
SuperGenPass	2.0	2.0	2.0	12.5	12.5	12.5
VuDroid	1.0	2.0	1.5	5.9	11.8	8.8
Average	2.2	3.5	2.8	9.8	14.4	11.9

Note: weight vector: (0.038, 0.051, −0.098, 0, −0.079, 0.194, 0.824).

Table 4. Results of Experiment 2

Application	Best	Worst	Med	Best (%)	Worst (%)	Avg (%)
Astrid	1.0	3.0	2.0	2.5	7.5	5.0
Omnidroid	1.0	3.0	2.0	5.6	16.7	11.1
Open Manager	2.0	2.0	2.0	7.7	7.7	7.7
Sipdroid	1.0	1.0	1.0	2.6	2.6	2.6
SuperGenPass	2.0	2.0	2.0	12.5	12.5	12.5
VLC	1.0	4.0	2.5	3.1	12.5	7.8
Average	1.1	2.1	1.6	4.9	8.5	6.7

Note: weight vector: (0.109, 0.067, −0.235, 0, −0.051, 0.275, −0.619).

Table 5. Results of Experiment 3

Application	Best	Worst	Med	Best (%)	Worst (%)	Avg (%)
APV	3.0	3.0	3.0	13.6	13.6	13.6
Astrid	1.0	3.0	2.0	2.5	7.5	5.0
KeePassDroid	1.0	7.0	3.0	3.0	21.2	10.9
Omnidroid	2.0	7.0	4.5	11.1	38.9	25.0
VLC	1.0	8.0	4.5	3.1	25.0	14.1
VuDroid	1.0	2.0	1.5	5.9	11.8	8.8
Average	1.5	5.0	3.1	6.5	19.7	12.9

Note: weight vector: (0.039, 0.062, 0, 0, −0.005, −0.011, 0).

Figs.12∼14 also show the ranked position of each leaking test case in the three experiments.

According to the results in Table 3∼Table 5, we can find that most of the test cases with memory leaks are ranked in the front of the prioritized list. On average, for the test cases with memory leaks, their best rank is 1∼2, their worst rank is 2∼5, and their median rank is 1∼3. This means the test cases with memory leaks can be tested at very first in our testing approach. The column `Med` roughly shows the distribution of the test cases with memory leaks in the prioritized results. We can see that the leaking test cases centralize in front of the ranked list. In most cases, the worst ranked leaking test cases occur in the first 25% of the prioritized test cases, and in more than half of the tested applications, a test case with memory leak can be ranked at the first position of the prioritized list. The column Avg (%) also shows that on average, the test cases with leaks can be ranked in the first 15% of the prioritized test cases. The above results indicate that our approach is effective in prioritizing memory leak test cases from a given test suite.

Because each test case needs to repeat event cycles for a lot of times to observe the memory usage, while the prioritization only needs to execute one event cycle in a test case, the prioritization cost is very small compared with fully executing these test cases, and the number of GUI events excluded from execution by prioritization can be large. This indicates the prioritization can help the users to speed up the memory leak testing process.

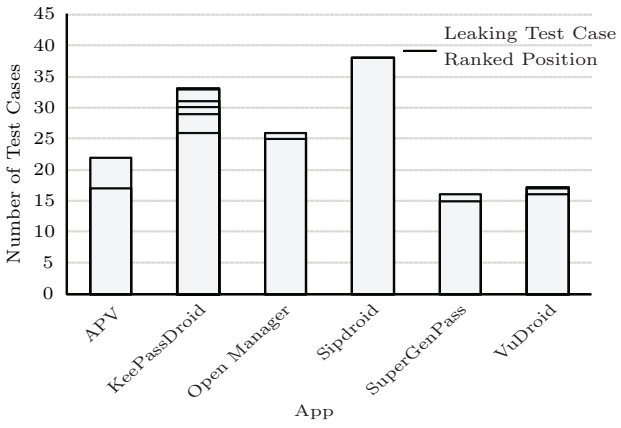


Fig.12. Ranks of the leaking test cases in experiment 1.

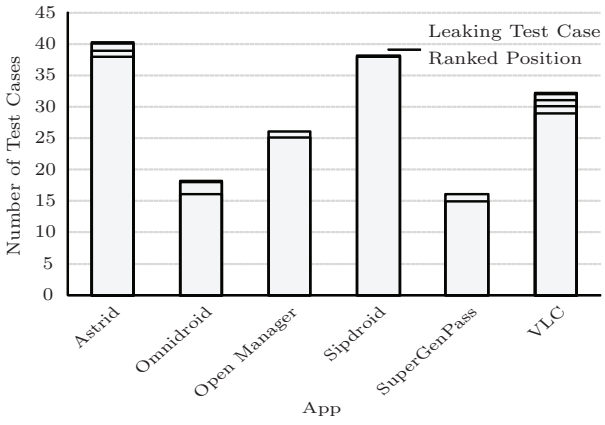


Fig.13. Ranks of the leaking test cases in experiment 2.

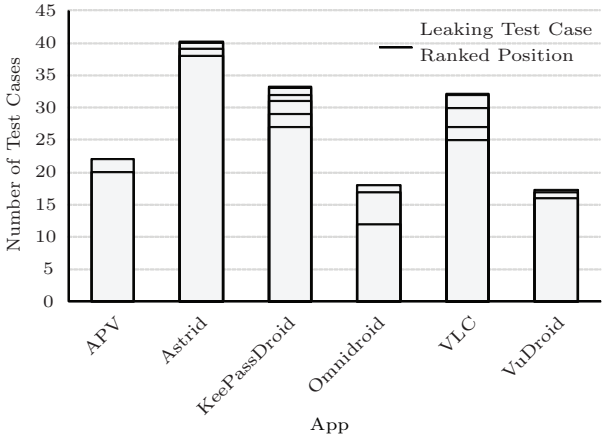


Fig.14. Ranks of the leaking test cases in experiment 3.

In Table 6, the average code feature vectors of the test cases with memory leaks and without memory leaks are shown, as well as the likelihood scores of memory leak produced by our leak prediction model. According to these experimental data, we can find the average scores of the test cases with memory leaks are much higher than those of the test cases without memory leaks. This means the score can indeed be an indicator of the likelihood of a test case to contain memory leaks.

6.3 Discussion

Although our leak relevant code features are not comprehensive, the experimental results show that they are sufficient for the studied subject programs. This suggests that for memory leak directed test case prioritization, we may not need to extract very comprehensive code features. Keeping the code feature vector simple not only works under many situations, but also can make the prioritization more lightweight.

By intuition, the weight for a code feature like the number of `onDestroy()` method calls is expected to be negative, and the weight for a code feature like the number of `getResources()` calls is expected to be positive. However, in the experiment, such weights are sometimes not as expected. This is mainly because the number of those calls is often very small. Therefore, during training, their weights are easily to be affected by other factors. However, the experiments show that such unexpected weights do not seriously affect the prioritization results.

In the experiment, we assign three leak levels {1, 2, 3} to the training test cases. Actually, according to the theory of the Ranking SVM algorithm, when more leak levels are assigned, the gap between the scores of the leaking test cases and the scores of the non-leaking test cases can be larger, and the prediction can possibly be more precise. However, determining what level to assign to each training test case is difficult. Therefore, this paper only uses three levels.

The threats to the validity of our experiment mostly exist in the number and the diversity of the benchmark applications we study. In the future, we plan to use more experimental subjects to further validate the effectiveness of the proposed approach. The evaluation will not only be conducted on normal user applications, but also be conducted on those launcher applications and games which tend to be resource-intensive. The size of the test suite and the number of the leaking test cases are another two threats. We currently mainly use the test cases provided in [13] for study. In the future, we plan to create more test cases and seek for more leaking executions to further optimize our memory leak prediction model.

7 Related Work

Memory Leaks in C/C++. Most of the existing research on C/C++ memory leaks focuses on static leak detection[21-24]. The detection methods can be based on escape analysis[21], shape analysis[22], ownership model[23], value-flow analysis[24], etc. With proper test cases, the leaks can also be detected by dynamic analysis using techniques similar to garbage collection[25]

Table 6. Average Feature Vector and Score Data

Application	Avg. Vector Data with Leak	Avg. Vector Data Without Leak	Avg. Score with Leak	Avg. Score Without Leak
APV	(223, 59, 99, 0, 15, 4, 0)	(63, 15, 29, 0, 1, 1, 0)	6.63	1.93
Astrid	(454, 78, 56, 0, 11, 16, 0)	(6, 6, 15, 0, 2, 8, 0)	34.06	4.06
KeePassDroid	(117, 66, 15, 0, 13, 1, 0)	(19, 6, 2, 0, 2, 0, 1)	7.33	1.15
Omnidroid	(501, 3, 24, 0, 82, 45, 0)	(284, 0, 62, 0, 127, 18, 0)	38.51	9.15
Open Manager	(116, 1, 0, 0, 0, 0)	(59, 0, 3, 0, 0, 0, 0)	8.60	3.39
Sipdroid	(222, 6, 2, 0, 0, 288, 0)	(8, 1, 0, 0, 0, 0, 0)	84.03	0.62
SuperGenPass	(32, 13, 2, 1, 0, 18, 0)	(14, 2, 0, 0, 0, 1, 0)	3.13	1.28
VLC	(138, 30, 24, 0, 23, 0, 1)	(57, 9, 60, 0, 2, 3, 1)	8.50	−2.23
VuDroid	(1 413, 341, 222, 0, 0, 4, 0)	(136, 20, 40, 0, 0, 0, 0)	63.32	4.35

or using object staleness profiling[26-27]. Besides detection, some other research also studies the validation[28] and the fixing[29] of C/C++ memory leaks.

Memory Leaks in Java and Other Managed-Languages. For managed-languages with garbage collection supports, the causes of leaks are more complex and hence the leaks are more difficult to detect[1]. Most of the existing studies in this area focus on leak diagnosis. They detect the root causes of leaks on a test execution likely to contain memory leaks. The diagnosing techniques can be roughly categorized into heap growth trends based ones[4-6], heap structure based ones[7-9], object staleness based ones[10-11], etc. In addition to the diagnosis, Pienaar and Hundt[2] and Yan *et al.*[3] also proposed static analysis based techniques to detect memory leaks in managed languages. They used special patterns in source code to reveal memory leak faults without needing a leaking execution.

Memory Leaks in Android Applications. Guo *et al.*[12] proposed an approach which uses static analysis to detect resource leaks in Android programs. The approach can find some memory leaks indirectly caused by unreleased resources; however, it cannot be used to detect general memory leak problem.

Because most of the existing work on memory leaks can only diagnose memory leaks while cannot discover leak phenomena in a systematic way, Yan *et al.*[13-14] proposed a systematic approach for testing resource leaks, including memory leak, for Android software. The approach is based on Android applications' GUI model. It generates comprehensive test cases to trigger repeated executions of neutral cycles to discover the memory leaks in applications. Although effective, the neutral cycle based approach may generate a large number of test cases for complex applications. Only a few of them can really lead to memory leaks. Therefore, testing all the generated test cases can be costly and unnecessary. Our prioritization technique can be viewed as an extension to Yan *et al.*'s work[13-14] to help further increase the test efficiency.

Shahriar *et al.*[15] also proposed a testing technique for Android memory leak problem. They did fuzz testing toward several identified memory leak patterns in Android applications. The problems revealed by their approach may not be real memory leak bugs and their testing is not comprehensive because of the limited number of patterns. Our leak relevant code features share some insights with their leak patterns. But different from Shahriar *et al.*'s work[15] which only considers Android framework related leak patterns, we also consider Java language level and Java runtime system level code features that are relevant to memory leaks. Therefore, our approach is not restricted in the Android platform and can also be extended to other systems.

8 Conclusions

In this paper, we proposed a novel approach that helps users prioritize test cases in a test suite for faster memory leaks detection. The approach firstly collects leak-relevant code features of a test case according to the use of Java language memory management instructions, Java runtime system resources, and application framework resources in a program. Then, we built a memory leak prediction model based on the extracted code features to determine the likelihood of a test case to contain leaks, and prioritize test cases according to their likelihood scores. The most suspicious test cases will be ranked to the front and suggested to the users in order to speed up the memory leak testing. Our evaluation on several freely available Android applications shows the proposed approach can prioritize test cases in high precision.

In the future, we plan to further extend our leak-relevant code features and the leak prediction model to make the proposed approach suitable for more applications. Besides prioritization, limiting the number of repetitions for event cycles in a test case can be another way to cut down test execution time. We also plan to study such approach in future work.

Acknowledgement We thank the anonymous reviewers for their insightful comments and criticisms.

References

[1] Šor V, Srirama S N. Memory leak detection in Java: Taxonomy and classification of approaches. *Journal of Systems and Software*, 2014, 96: 139-151.

[2] Pienaar J A, Hundt R. JSWhiz: Static analysis for JavaScript memory leaks. In *Proc. the IEEE/ACM International Symposium on Code Generation and Optimization* (*CGO*), February 2013, pp.11:1-11:11.

[3] Yan D, Xu G, Yang S, Rountev A. LeakChecker: Practical static memory leak detection for managed languages. In *Proc. the IEEE/ACM International Symposium on Code Generation and Optimization* (*CGO*), February 2014, pp.87-97.

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

[5] De Pauw W, Sevitsky G. Visualizing reference patterns for solving memory leaks in Java. In *Proc. the 13th European Conference on Object-Oriented Programming* (*ECOOP*), June 1999, pp.116-134.

[6] Xu G, Bond M D, Qin F, Rountev A. LeakChaser: Helping programmers narrow down causes of memory leaks. In *Proc. the ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI*), June 2011, pp.270-282.

[7] Rayside D, Mendel L. Object ownership profiling: A technique for finding and fixing memory leaks. In *Proc. the 22nd IEEE/ACM International Conference on Automated Software Engineering* (*ASE*), November 2007, pp.194-203.

[8] Mitchell N, Sevitsky G. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *Proc. the 17th European Conference on Object-Oriented Computing* (*ECOOP*), July 2003, pp.351-377.

[9] Maxwell E K, Back G, Ramakrishnan N. Diagnosing memory leaks using graph mining on heap dumps. In *Proc. the 16th International Conference on Knowledge Discovery and Data Mining* (*KDD*), July 2010, pp.115-124.

[10] Bond M D, McKinley K S. Bell: Bit-encoding online memory leak detection. In *Proc. the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS*), October 2006, pp.61-72.

[11] Xu G, Rountev A. Precise memory leak detection for Java software using container profiling. *ACM Transactions on Software Engineering and Methodology* (*TOSEM*), 2013, 22(3): Article No. 17.

[12] Guo C, Zhang J, Yan J, Zhang Z, Zhang Y. Characterizing and detecting resource leaks in Android applications. In *Proc. the 28th IEEE/ACM International Conference on Automated Software Engineering* (*ASE*), November 2013, pp.389-398.

[13] Yan D, Yang S, Rountev A. Systematic testing for resource leaks in Android applications. In *Proc. the 24th IEEE International Symposium on Software Reliability Engineering* (*ISSRE*), November 2013, pp.411-420.

[14] Zhang H, Wu H, Rountev A. Automated test generation for detection of leaks in Android applications. In *Proc. the 11th IEEE/ACM International Workshop on Automation of Software Test* (*AST*), May 2016, pp.64-70.

[15] Shahriar H, North S, Mawangi E. Testing of memory leak in Android applications. In *Proc. the 15th IEEE International Symposium on High-Assurance Systems Engineering* (*HASE*), January 2014, pp.176-183.

[16] Park J, Choi B. Automated memory leakage detection in Android based systems. *International Journal of Control and Automation*, 2012, 5(2): 35-42

[17] Vallée-Rai R, Co P, Gaghon E *et al.* Soot – A Java optimization framework. In *Proc. the IBM Centre for Advanced Studies Conference* (*CASCON*), November 1999, Article No. 13.

[18] Joachims T. Optimizing search engines using ClickThrough data. In *Proc. the ACM Conference on Knowledge Discovery and Data Mining* (*KDD*), July 2002, pp.133-142.

[19] Yang W, Prasad M, Xie T. A grey-box approach for automated GUI-model generation of mobile applications. In *Proc. the International Conference on Fundamental Approaches to Software Engineering* (*FASE*), March 2013, pp.250-265.

[20] Zhang P, Elbaum S. Amplifying tests to validate exception handling code. In *Proc. the 34th International Conference on Software Engineering* (*ICSE*), June 2012, pp.595-605.

[21] Xie Y, Aiken A. Context- and path-sensitive memory leak detection. In *Proc. the 10th European Software Engineering Conference Held Jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, September 2005, pp.115-125.

[22] Dor N, Rodeh M, Sagiv S. Checking cleanness in linked lists. In *Proc. the International Static Analysis Symposium* (*SAS*), June 29-July 1, 2000, pp.115-134

[23] Heine D L, Lam M S. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proc. the ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI*), June 2003, pp.168-181.

[24] Sui Y, Ye D, Xue J. Static memory leak detection using full-sparse value-flow analysis. In *Proc. the International Symposium on Software Testing and Analysis* (*ISSTA*), July 2012, pp.254-264.

[25] Hastings R, Joyce B. Purify: Fast detection of memory leaks and access errors. In *Proc. the Winter USENIX Conference*, January 1992, pp.125-138

[26] Jung C, Lee S, Raman E *et al.* Automated memory leak detection for production use. In *Proc. the 36th International Conference on Software Engineering* (*ICSE*), May 31-June 7, 2014, pp.825-836.

[27] Lee S, Jung C, Pande S. Detecting memory leaks through introspective dynamic behavior modelling using machine learning. In *Proc. the 36th International Conference on Software Engineering*, May 31-June 7, 2014, pp.814-824.

[28] Li M, Chen Y, Wang L, Xu G. Dynamically validating static memory leak warnings. In *Proc. the ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA*), July 2013, pp.112-122.

[29] Gao Q, Xiong Y, Mi Y, Zhang L, Yang W, Zhou Z, Xie B, Mei H. Safe memory-leak fixing for C programs. In *Proc. the 37th International Conference on Software Engineering* (*ICSE*), May 2015, pp.459-470.

Ju Qian received his B.S. and Ph.D. degrees in computer science from Southeast University, Nanjing, in 2003 and 2009, respectively. He is currently an associate professor in the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing. His research interests include program analysis, software testing, debugging, and evolution.

Di Zhou received her B.S. degree in computer science from Nanjing University of Aeronautics and Astronautics, Nanjing, in 2013. She is currently pursuing her M.S. degree in the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing. Her research mainly focuses on software testing.