CS6880 Software Engineering

Light-weight Dead Code Analysis for Precise Code Coverage in Android Apps

Gao Xiang, A0159469J
19, Nov, 2016

Abstract

Dead code in real-life applications may reduce the code coverage in program analysis. There are some existing tools to detect the dead code in java program. Though android apps are written in java language, they have some special properties: Android apps do not have a main method, communication between different app components is done means of messages. Those properties make it highly likely that such tools may have higher false positive rate when used to Android apps. So, in this project, I have implemented a light-weight dead code detection tool for android apps. Experimental results show that this tool can detect some dead codes that UCDetector[1] and ProGaurd[2] cannot find.

Introduction

In computer programming, dead code is a section in the source code of a program which can never be accessed, or can be executed but whose result is never used in any other computation. The execution of dead code wastes computation time and memory. While the result of a dead computation may never be used, it may raise exceptions or affect some global state, thus removal of such code may be good for program performance and reduce the misleading in code coverage. Dead code can be eliminated by static code analysis and data flow analysis.

Different from traditional programs, android programs do not have a unique main method. And android apps mainly contain 4 components: Activity, Service, Broadcast Receiver and Content Provider. The communication between different app components is done by message - Intent. Determining the inter-component interaction may be non-trivial. Those indirect invocation paths bring some challenges to dead code detection. The possible solution may involve statically analysing the app byte code to find out all the activities that can be reached from the start activity of an app. On the other hand, the xml file in an android app may also contain some program information. So we have to consider not only the java file, but also the xml file.

Abstract syntax tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. AST can be used to static analysis of android program, which provides a convenient method to travel the source code.

In this project, I first build a direct call graph using an existing tool – AndroGuard. Secondly, adding the indirect call edge, for instance intent and button registration, using AST. Thirdly, get the main activity from manifest.xml file. And finally, travel from the method invoked by system in the main activity along with the call graph, the method that cannot be reached will be regarded as dead code.

Existing Tools

AndroGuard[3]

Androguard is mainly a tool written in python to play with: * Dex/Odex, * APK, *Android's binary xml, * Android Resources. This tool can be used to de static analysis for android apps, which can build control flow graph, call graph and so on. In this project, I used AndroGuard to build the incomplete call graph. The call graph built by Androguard does not contain the indirect edge, such as Intent.

UCDetector

UCDetector (Unnecessary Code Detector) is an eclipse PlugIn tool to find unnecessary (dead) public java code. For example, public classes, methods or fields which have no references. But I find that UCDetector detects dead code just according to the references, which could cause some false negatives. For example, if I declare an intent(activity_a, activity_b) which indicates the active activity will change to activity_b from activity_a when startActivity(intent) is invoked. But if there is not statement invoke startActivity(intent), activity_b will be a dead code. However, from the perspective of UCDetector, there is also a reference to activity_b, and it will not regard activity_b as dead code.

ProGuard

ProGuard is a free Java class file shrinker, optimizer and obfuscator. It detects and removes unused classes, fields, methods, and attributes. It optimizes bytecode and removes unused instructions. It renames the remaining classes, fields, and methods using short meaningless names. The resulting applications and libraries are smaller, faster, and a bit better hardened against reverse engineering. According to my observation, ProGuard detects dead code just according to the references, which will cause some false negative just like UCDetector.

Implementation

In this section, I will explain my implementation of my tool in detail. There are mainly 3 steps in the implementation: building call graph, determining the main activity and determining the dead codes.

Building Call Graph

Just as mentioned above, AndroGuard can be used to build call graph for android apps. But the call graph built by this tool does not contain the indirect edge. In this part, I used AndroGuard to get all the direct relationship of each method, and then added indirect edge using static analysis.

To get the indirect relationship, I just concentrate on Intent and callback registration. In order to conduct a static analysis, I build an Abstract syntax tree (AST) for each java file first, which provides a simple way to travel the source code. Then, monitoring the invocation of startActivity(intent). The source and target of the intent will be parsed when such a method is invoked. In this situation, an indirect edge from source to target will be added to the call graph. In a similar way, when a setOnClickListener(listener) function is invoked, an edge from current

method to the listener.onClick will be added. Besides, we should build an edge from Thread.start() to run() function, which also can be regarded as an indirect edge.

Determining the Main Activity

The main activity is the first activity when an app is started. Actually, the main activity can be regarded as the entry point of android apps. However, there may be several entry points in the activity, for example, onCreate(), onResume(), onPause(), onDestroy() and so on. Those methods can be invoked by system.

The main activity information can be extracted from xml file. In the manifest.xml, the activities containing attribute “intent-filter” is the main activities. So I parse the manifest.xml file to get the main activity information.

Meanwhile, there are some xml files may invoke java method. For example, the layour.xml file may contain Textview or Button defined in java files. In this situation the constructor and event-handler function in java files can be invoked by system. Because there are so many widgets in android API, I just implement the TextView.

Determining the Dead Codes

After build the call graph and determine the main activity, determining the dead codes becomes a much easier task. In my implement, I travel the call graph from all the entry points in the main activity. The method that cannot be accessed will be regarded as dead code.

Evaluation

Benchmark

In this project, I used ChessClock as my benchmark. Because there is no dead code in the original code, I added some method that will not be invoked, and modified some piece of code to produce dead codes. The detail of my modifications are as follows:

- removing the statement “startActivity(prefsActivity)” in ChessClock.java(line 1004).
- removing the statement “pause.setOnClickListener(P2ClickHandler)” in ChessClock.java(line 628)
- add a method deadCode() in the end of the ChessClock.java

Test Cases

click p1(the button on top of the screen) – click pause – click start – click menu – click setting – click Reset Clocks – click yes -- click Reset Clocks – click no -- click about -- back

Code Coverage

The code coverage for CheeClock using above test case is as follows:

EMMA Coverage Report (generated Sat Nov 19 14:43:31 SGT 2016)

[all classes]

COVERAGE SUMMARY FOR PACKAGE [com.chessclock.android]

name	class, %	method, %	block, %	line, %
com.chessclock.android	92% (12/13)	62% (46/74)	49% (1026/2113)	51% (232.6/452)

COVERAGE BREAKDOWN BY SOURCE FILE

name	class, %	method, %	block, %	line, %
Prefs.java	0% (0/1)	0% (0/2)	0% (0/18)	0% (0/6)
ChessClock.java	100% (10/10)	63% (42/67)	47% (956/2021)	50% (215.6/427)
CTextView.java	100% (1/1)	67% (2/3)	89% (31/35)	82% (9/11)
DialogFactory.java	100% (1/1)	100% (2/2)	100% (39/39)	100% (8/8)

Fig 1: Code Coverage with Dead Code

From Fig 1, we can find that the coverage in class level is 92%. There is 1 class that was not accessed, because we have remove startActivity(intent) which will invoke prefs.java. So prefs.java is a dead code.

In method level, the coverage is 62%. Because I have removed the statement "pause.setOnClickListener (P2ClickHandler)", the handler function will not be invoked, that is, the handle function is a piece of dead code. Meanwhile, my test case did not cover all the method that can be invoked.

As for line level, we can find that the coverage is just 51%.

Dead code may have a great effect on code coverage. Fig 2 shows the coverage after removing all the dead code.

EMMA Coverage Report (generated Sat Nov 19 14:56:46 SGT 2016)

[all classes]

COVERAGE SUMMARY FOR PACKAGE [com.chessclock.android]

name	class, %	method, %	block, %	line, %
com.chessclock.android	100% (11/11)	66% (45/68)	53% (1008/1912)	57% (230.6/406)

COVERAGE BREAKDOWN BY SOURCE FILE

name	class, %	method, %	block, %	line, %
ChessClock.java	100% (9/9)	65% (41/63)	51% (938/1838)	55% (213.6/387)
CTextView.java	100% (1/1)	67% (2/3)	89% (31/35)	82% (9/11)
DialogFactory.java	100% (1/1)	100% (2/2)	100% (39/39)	100% (8/8)

Fig 2: Code Coverage without Dead Code

We can find that the test case has covered all the classes. Method coverage becomes 66% from 62%, and the line coverage is increased by 6%.

Compare with Other Tools

Fig3 shows my execution results. We can find that my tool can find four pieces of dead code. The first one is the handler function of P2ClickHandler. The second one is a function that is invoked in the handler function of P2ClickHandler. And the third one is the deadCode() function I added. And the last one is the onCreate function in Prefs.java which should be invoked by intent in original program.

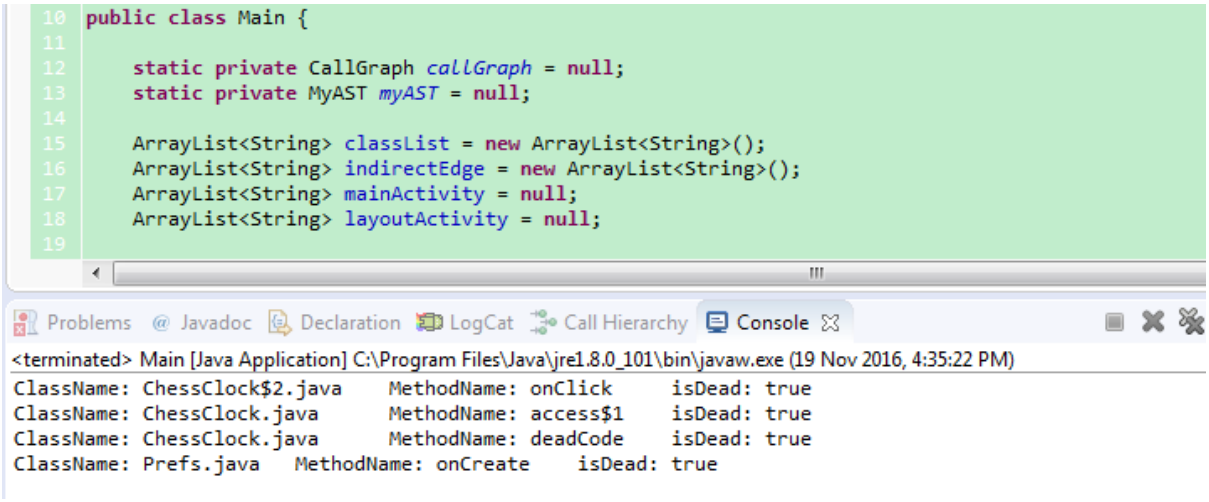

Fig 3: Results of My Tool

And Fig 4 shows the results of DCDetector. Actually, DCDetector found many methods and classes that should change the visibility. As for dead code, it just found one method deadCode() has 0 references. That is, DCDetector regards deadCode() function added by me as dead code.

Nr	Java	Marker	Description	References**
1	Method		Change visibility of Method "ChessClock.CheckForNewPrefs()" to private	-
2	Method		Method "ChessClock.deadCode()" has 0 references	0
3	Method		Change visibility of Method "ChessClock.PauseToggle()" to private	-
4	Constant		Change visibility of Constant "ChessClock.TAG" to private	-
5	Constant		Change visibility of Constant "ChessClock.V_MAJOR" to private	-
6	Constant		Change visibility of Constant "ChessClock.V_MINOR" to private	-
7	Constant		Change visibility of Constant "ChessClock.V_MINI" to private	-
8	Field		Change visibility of Field "ChessClock.P1ClickHandler" to private	-
9	Field		Change visibility of Field "ChessClock.PauseListener" to private	-
10	Class		Change visibility of Class "DialogFactory" to default - May cause compile errors!	-
11	Method		Change visibility of Method "DialogFactory.AboutDialog(Context,String,String,String)" to default	-
12	Class		Change visibility of Class "Prefs" to default - May cause compile errors!	-
13	Interface		Change visibility of Interface "FinishListener" to default - May cause compile errors!	-
14	Class		Change visibility of Class "EmmaInstrumentation" to default - May cause compile errors!	-
15	Field		Change visibility of Field "EmmaInstrumentation.TAG" to private	-
16	Class		Change visibility of Class "InstrumentedActivity" to default - May cause compile errors!	-
17	Field		Change visibility of Field "InstrumentedActivity.TAG" to private	-

Fig 4: Results of DCDetector

I also try to use proGuard to detect the dead code. The results show ProGuard did not find any dead code. But, ProGuard can optimize some pieces of code, and remove the unnecessary code. For example, if we declare a final field, whose value will not be modified any more. ProGuard will remove the declared variable, and use the value directory in the following codes.

Discussion and Conclusion

Actually, this is just a light-weight implementation. This tool can detect dead code in method level. As for the dead code in the method, this tool cannot find them. This kind of dead code can be detected by build a data dependency graph, the statements that can be executed but whose result is never used in any other computation can be regarded as dead code. Meanwhile, we can detect the dead code inside a method by build a control flow graph. The statement which can never be reached will be regarded as dead codes.

In conclusion, I have considered the properties of android apps, and built a call graph including the indirect call edge using AST. To extract the information in the xml file, I have also done a simple analysis for the xml file. The simple tool I implemented can find many dead codes that previous tools cannot find. However, this tool just can be used to detect the dead code in method level. The dead code inside the method cannot be found by this method.

References

[1] http://www.ucdetector.org/

[2] http://proguard.sourceforge.net/

[3] https://code.google.com/archive/p/androguard/