Dead Code Elimination

Removal of unreachable methods and dead if conditions in Android applications

Vaishnavi Mohan
Technische Universität Darmstadt
Germany
vaishnavi.mohan@stud.tu-darmstadt.de

ABSTRACT

There are many applications with dead code in them. The presence of such dead and unreachable code may degrade the performance of the applications. This paper describes a method to statically analyse Android application code and detect such dead code. The paper also describes how such dead code can be removed from the application. The results of the analysis show that about 1/3rd of Android applications have some dead code in them.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

ACM proceedings, LaTeX, text tagging

1. INTRODUCTION

According to Wikipedia, unreachable code is part of the source code of a program which can never be executed because there exists no control flow path to the code from the rest of the program. Dead code or Unreachable Code maybe present in the program due to many reasons including debugging constructs used by the programmer, method definitions in the programs without calls to them and conditional branches that are never executed due to the input. Such dead code is unwelcome because it may increase the memory requirements of the applications, it may even make static analysis imprecise as explained in the example below and it may even cause slower static analysis of the application since time is wasted analysing undesirable code.

```
void leak(){
String taint="secret";
int i=0;
if(i>0)
send(taint);
else
print("No taint");
}
```

The example above illustrates how dead code may inject false positives in static taint analysis. When taint analysis is performed on the method `leak()`, a leak of secret information is reported by the analyser, though in practice, the `send(taint)` method is never invoked as the if condition never evaluates to true. Removing this dead code will ensure that the taint analysis does not report these kind of false positives.

The Soot framework aids in the analysis, instrumentation, optimization and visualization of Java and Android applications. The Soot framework already provides a partial dead-code elimination implementation. The implementation is partial in the sense, only intra-procedural dead code is detected and eliminated. In this project, inter-procedural dead code elimination has been implemented. The implementation detects and eliminated inter- and intra-procedural dead code using constant propagation with the help of the VASCO framework. The VASCO framework performs precise inter-procedural data flow analysis using value sensitive contexts.

The project implements two kinds of dead-code elimination. Methods that are defined in the application but never invoked are detected as unused methods and are eliminated from the application. Secondly, if statements that always evaluate to true or always evaluate to false are detected and the appropriate part of the if statement that is not executed is eliminated.

The rest of the report is organised as follows: the next section describes how unused methods are detected and eliminated in this project, Section 2 illustrates how the dead if statements were detected and eliminated, Section 3 describes some results of our analysis and Section 4 describes my contribution to the project, and finally the paper ends with the derived conclusion and score for future work.

2. UNUSED METHOD REMOVAL

To detect the methods residing in the application but not invoked, we make used of a callgraph. The Soot framework provides the facilities to build a callgraph that models the control flow in the program being analysed. Soot provides many variations of such callgraphs. The implementation to remove dead methods makes use of the SPARK callgraph due to its precision. The callgraph contains edges modeling the control flow in the program. We now get the list of unused methods in the program by comparing the methods present in the callgraph with all the methods present in the application. The unused methods that were detected are then deleted from the application.

3. DEAD-IF REMOVAL

To detect dead-if conditions, we make use of constant propagation in the VASCO framework. When an if condition is encountered, the analysis checks if the evaluation of the condition is a constant value through all possible propagations of the program.

```
void detect(){
int i=0,j=1;
if(i==0 and j==1)
print("good code");
else
print("bad code");
double r=Math.random();
if(r==444)
print("hello");
else
print("bye");
}
```

For example, in the above code, when the first if condition is encountered, the values of the variables i and j are checked in the preceding statement (called as predecessor) to the if condition. Here the predecessor is the declaration statement of the variables, and here since the if has only one predecessor we know that the value of i and j are always 0 and 1 respectively and that the condition evaluates to true always. Here, we report that this if condition always evaluates to true and that the else part can be removed. In the case of the next if statement, the value of r is a random number and not a constant. Hence, this if statement is not reported as having a constant evaluated condition. When the if statement's condition always evaluates to true, we remove the if statement and replace it with the goto statment that is part of the if statement. In the above example, we remove the if(i==0 and j==1) and replace it with a goto statement that points to the statement print("good code"). Now, since there is no if statement, the else part has no call to it and can be marked as unreachable code which can then be removed. When the if statement's condition always evaluates to false, we remove the if statement along with the goto stament. We then run Soot's unreachable code eliminator to remove the parts to which now no goto statement exists.

4. CONTRIBUTIONS

In this project, I worked on detection and removal of unreachable methods in the program. To implement this, we first needed to create a callgraph to model our application. Java has a main method and from this the callgraph can be modelled. But, in Android, we have no main method and hence, need to create a dummy main method to give it as a the entrypoint for callgraph generation.

To create a dummymain method that has all the callbacks in the Android application, we made use of FlowDroid's createDummyMain() method in the BaseEntryPointCreator class. This dummy main models all the possible entry points to the android application based on the Android application lifecycle.

The callgraph is then generated with the entrypoint as this dummy main function. The Spark callgraph is used. The callgraph contains edges to all the methods that are actually called in the execution of the program. The methods in tha application are retrieved from the Scene using the getApplicationClasses() function, from which all methods are then retrieved from each class. Now this list of methods is compared with the list of methods retrieved from the callgraph present in the scene. The methods that are present in the application but not in the callgraph are marked as unused methods. The unused methods are then removed from their respective classes using the function class.removeMethod().

The callgraph did not model the R.java files in android and hence, the init() methods of the R class were being deleted from the application and this created issues in running the android application after analysis. Since, the init() methods of the R.java files should not be removed from the application, we handled an explicit case that the init() of R classes should never be removed from the application.

For dynamic invocation of methods, that is reflection, the callgraph does not have edges to these methods. Hence, we have partially handled this, by not removing the methods that are passed as a string to the method.invoke() function. Ideally, we should add these methods to the callgraph and recompute the callgraph.

5. CONCLUSION

We have tested our implementation on 18 android applications, out of which 6 of them had dead code in them. On dead code removal in these applications, we found that there is a decrease in the size of the application's size. We also found that all of these 6 applications contained unused methods whereas only 33

5.1 References

www.wikepedia.org FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps http://sable.github.io/soot/ https://github.com/roh