VOL. 11, NO. 22, NOVEMBER 2016 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences

www.arpnjournals.com

MOBILE ANALYZER: AN ANALYSIS TOOL FOR ANDROID APPS

Maryam Ahmed, Rosziati Ibrahim and Noraini Ibrahim
Department of Software Engineering, FSKTM, Universiti Tun Hussein Onn Malaysia, Malaysia
E-Mail: hi120055@siswa.uthm.edu.my

ABSTRACT

Android applications continue to dominate the mobile market as time passes. However, software quality of the application remains a challenge. Analysis of the android application has been done by different researchers in the area of security, power consumption and performance of the android application. Source code refactoring has been used in android application to improve power consumption and execution time. In the area of testing that certify the quality of the product to continue keeping the app in the market place, analysis of the application is yet to be explored and research is yet to cover refactoring to improve android application testing. This paper proposes an analysis tool that improves the testing process of the android application. Our analysis tool extends the ApkAnalyzer and refactors the bytecodes generated to reduce the test path thus test cases generated are limited and have efficient coverage.

Keywords: mobile analyzer, android analyzer, refactoring, mobile testing.

INTRODUCTION

Android is an open source software development platform that runs on various mobile smart devices (Ko *et al*., 2012). Being the first free open source and fully customizable software for mobiles, it cuts across different hardware manufacturers including Sony, Samsung, HTC, etc. (Asaithambi and Jarzabek, 2013). Android software includes the operating system OS, middleware and major applications running on mobile (Asaithambi and Jarzabek, 2013) (Ko *et al*., 2012). According to latest information from international data cooperation (IDC), Android broke through the mobile market with over one billion unit mark in 2014, a noteworthy landmark in the progressive record of the mobile market. This profound success has been attributed to the strong end user demands, refreshed product portfolios and availability of low-priced mobile phones that can run on android OS (IDC, 2015). To maintain this achievement, software quality of the applications that run on android need to be certified. Reliability of the android application lies on the quality assurance unit. Software reliability can be enhanced by applying improved testing techniques and policies (Machado, Campos and Abreu, 2013). Test case generation is an important aspect of software testing. Test cases generated from SOFTWARE UNDER TEST (SUT) can represent the working system of the SOFTWARE UNDER TEST (SUT) (Asaithambi and Jarzabek, 2013). However, study shows high rate of redundancies in android application hence test case (Asaithambi and Jarzabek, 2013).To further comprehend the redundancy in android, analysis of the source code needs to be done. The software development kit of android is predominantly based on a popular programming language, Java. This has enhanced the creation and recreation of more android programs but with limited number of quality applications (Kraemer, 2011).

Refactoring has been used over time to improve software development. Refactoring is a code transformation procedure that affects only the structure of the code without adding or removing any functional feature of the software (Alves *et al*., 2013). The code transformation has been focusing on improving the software quality by making the source code more readable and easy to maintain. Study has shown that almost all android applications need enhancement through refactoring (Zhang *et al*., 2012). The process of refactoring ranges from simple name editing to more complex changes to satisfy the purpose of transformation and it has been applied at both source code and bytecode level. A research (Zhang *et al*., 2012) has it that refactoring at bytecode level is more efficient than the java code in terms of execution time and energy consumption with 46-97% faster in execution time and energy consumption reduced by 27-83%, hence emphasizing bytecode refactoring.

Several android analyser framework and tool has been designed and developed, especially in area of security to detect vulnerabilities. However, researchers are yet to explore analysis tool that aim at functional testing, while considering one of google best practices of avoiding creation or existence of unnecessary objects due to the cost of maintenance.

This work proposes an analysis tool for android using the android ApkAnalyzer. To eliminate unwanted objects in the software, we are refactoring the bytecode to give more precise test cases with complete coverage. The remaining section in this paper is organized as follows. Second II discusses the related work on analysis and refactoring. ApkAnalyzer used in our proposed model is discussed in section III while section IV presents our approach. A case study is presented in section V.

RELATED WORK

Android remain an active area of research due to its market coverage. Several analysis tool has been developed over time (Grace *et al*., 2012) (Vekris et. al., 2012) (Yang *et al*., 2013) especially in the area of android security.

In (Grace *et al*., 2012), the woodpecker system is presented to examine how permission based security model is enforced by employing interprocedural data flow analysis to systematically expose possible leaks an unauthorized user can access sensitive data. Similar to

(Grace *et al*., 2012), (Vekris *et al*., 2012) conducted their research on how to detect leakage in android apps using the AppIntent to analyse data transmission intended by the user. In (Yang *et al*., 2013), android applications are analysed to verify which of the applications abide by a set of policies.

Refactoring on the other hand has been widely used in improving software development including android applications (Zhang *et al*., 2012) (Moghadam and Cinneide, 2012) (Hecht *et al*., 2015).

A tool named DPartner was presented (Ying *et al*., 2012) that automatically refactors android application with computation offloading capability by analyzing its bytecode to discover where offloading is necessary and then rewrites the bytecode. D Partner evaluates three android application to discover reduction of execution time by 46%-97%. Iman and Mel presented a novel approach that helps reduce maintainance effort by automatically refactoring source code to achieve a high quality design version of the current system. Another tool Paprika (Geoffrey *et al*., 2015) is proposed to analyze android application to detect android specific antipatterns from binaries of mobile applications.

APKANALYZER

ApkAnalyser is a static, virtual analysis tool, which can be used to analyse API references, view application architecture and dependencies, and disassemble bytecodes in Android apps (Sony, 2015). It's a complete tool chain which supports modification of the binary application with more printouts. The modified binary code (bytecode) can then be able to repack, install, run and verify the result from the logcat. Some of the features of the ApkAnalyzer include:

It explores and lookup packages, classes, methods and fields.

It can disassemble the Dalvik bytecode method keeping the syntax highlighted.

It can also decode the Android XML file with highlighted syntax.

It displays UML packages and class diagrams, and highlights class dependencies.

It modifies the APK file with predefined Dalvik bytecode injections due to changes made to the bytecode.

Generally, there are four kinds of output in the display window of the ApkAnalyzer:

a) UML diagrams for packages and classes.

b) A Dalvik disassembler for methods.

c) A resource detail view for resource IDs.

d) An XML decoder for XML resources.

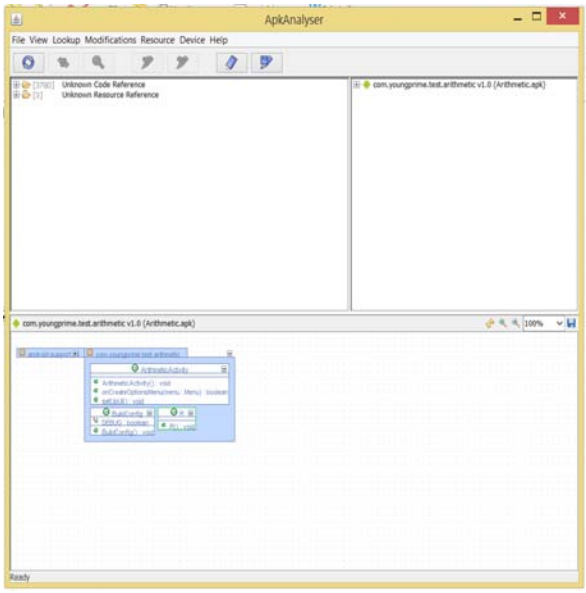

Figure-1. An overview of ApkAnalyzer display.

Another good use for ApkAnalyser is the ability to create a set of bytecode modifications, which could be applied to the APK file in a batch. This automatically adds printouts of suspicious pieces of code, to support in investigating the execution flow of the application.

MOBILE_ANALYZER

The proposed tool is a fragment of our previously proposed testing model. The proposed testing model is an adaptation model from a particular web application testing model (Reweb and Testweb) (Ahmed and Ibrahim, 2015). The testing model has two parts as the adapted model; Mobile_Analyzer and Test_Mobile. The Mobile_Analyzer has as its input the android apk by setting the path of the middle to the apk of the SOFTWARE UNDER TEST (SUT) on the X_ApkAnalyzer. The X_ApkAnalyzer is an extension of the open source ApkAnalyzer. The X_ApkAnalyzeranalyse the apk to generate the bytecode. We then apply one of google best practices of eliminating unnecessary objects by refactoring the bytecode to generate more precise bytecode for further analysis. Further analysis is done to generate the UML model which will be later use for test case generation.

Refactoring is done based on the purpose for refactor. The purpose of refactoring in this case is to reduce test cases generated while maintaining a good coverage. Our first tactic is to minimise dependencies between classes hence making the codes more testable with reduced number of test paths. The next step is to identify repetitive methods on same objects in different classes of the code, then reconstruct the code while keeping all functionalities unbroken. The following steps are followed to refactor:

a) Identify all classes in the set Ciof the source code of the software under test.

b) Create a state S(Ci) for each class in the universal set Ci.
c) Identify the relationship between each class in the set.
d) Identify all test paths P in the program.
e) Identify unworkable test path and the respective classes.
f) Reconstruct the bytecode to eliminate redundant classes.
g) Identify set of new classes NC of the code.
h) For each class NCi of the new code, check for repetitive methods.
i) Reconstruct to reduce repetitive testing of same object in different classes.
j) Run the new code to ensure all functionalities are not altered.
k) Validate the new bytecode to generate the new UML model.

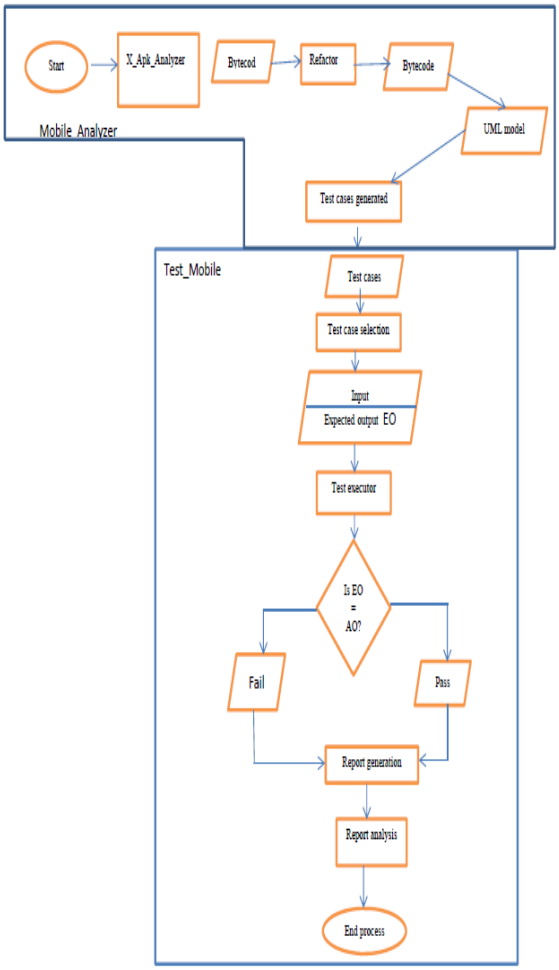

Figure-2. Adaptation testing model.

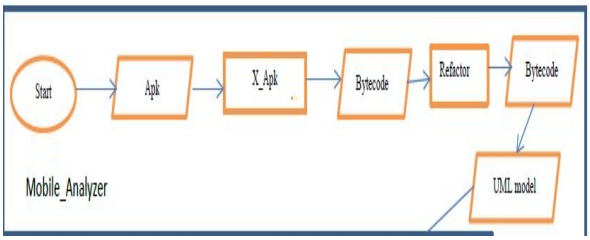

Figure-3. Mobile_analyzer framework.

CASE STUDY

This section gives more insight into our approach by presenting a case study of a regenerated bytecode with no alteration in it functionality and keeping the test path to the best minimal using the technique explained in section IV. The apk is loaded on the X_ApkAnalyzer to generate the bytecode. For the purpose of this study, the SOFTWARE UNDER TEST (SUT) has one main class and is valid for purpose of testing; hence, no class was eliminated. However, in a class, there's repetitive method and this was handled by reconstructing the bytecode to avoid a repetitive task in process of testing. The new bytecode is then generated. To verify the validity of our approach, the two bytecode is used to generate the source code. The source code is then run using the emulator in the Android SDK. The emulator is configured to run on Android 4.4 API 19, simulating the ARM (armeabi-v7a) processor.

From Figure-4(a), there is acommon call of IF statement, which is called under each function in (a). In (b), it is tested once outside the methods, hence reducing the number of times the numbers are tested for null. The change applied has no effect on their output as shown in figure.

```
private void addNumbers(){
        String num1 = numberOne.getText().toString();
        String num2 = numberTwo.getText().toString();
        if(num1 != null && num2 != null){
                dblNum1 = Double.parseDouble(num1);
                dblNum2 = Double.parseDouble(num2);

                double dblResult = dblNum1 + dblNum2;
                result.setText("Result : " + dblResult);
        }
}
private void subtractNumbers(){
        String num1 = numberOne.getText().toString();
        String num2 = numberTwo.getText().toString();
        if(num1 != null && num2 != null){
                dblNum1 = Double.parseDouble(num1);
                dblNum2 = Double.parseDouble(num2);

                double dblResult = dblNum1 - dblNum2;
                result.setText("Result : " + dblResult);
        }
}

private void multiplyNumbers(){
        String num1 = numberOne.getText().toString();
        String num2 = numberTwo.getText().toString();
        if(num1 != null && num2 != null){
                dblNum1 = Double.parseDouble(num1);
                dblNum2 = Double.parseDouble(num2);

                double dblResult = dblNum1 * dblNum2;
                result.setText("Result : " + dblResult);
        }
}

private void divideNumbers(){
        String num1 = numberOne.getText().toString();
        String num2 = numberTwo.getText().toString();
        if(num1 != null && num2 != null){
                dblNum1 = Double.parseDouble(num1);
                dblNum2 = Double.parseDouble(num2);

                double dblResult = dblNum1 / dblNum2;
                result.setText("Result : " + dblResult);
        }
```

Figure-4(a). Original code.

```
String num1 = numberOne.getText().toString();
        String num2 = numberTwo.getText().toString();
        if(num1 != null && num2 != null){
                dblNum1 = Double.parseDouble(num1);
                dblNum2 = Double.parseDouble(num2);


private void addNumbers(){

                double dblResult = dblNum1 + dblNum2;
                result.setText("Result : " + dblResult);
        }
}
private void subtractNumbers(){

                double dblResult = dblNum1 - dblNum2;
                result.setText("Result : " + dblResult);
        }
}

private void multiplyNumbers(){

                double dblResult = dblNum1 * dblNum2;
                result.setText("Result : " + dblResult);
        }
}

private void divideNumbers(){

                double dblResult = dblNum1 / dblNum2;
                result.setText("Result : " + dblResult);
        }
```

Figure-4(b). Refactored code.

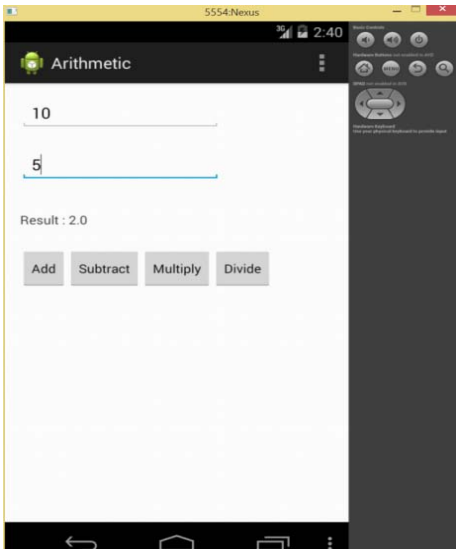

Figure-5(a). Unaltered sourcecode output.

Figure-5(b). Altered sourcecode output.

CONCLUSIONS

This paper presents Mobile_Analyzer, an analysis tool for android application to improve android application testing by applying android best practices of removing unused classes and reducing repetitive object use within the code. A case study is presented to illustrate our approach. The apk is analysed and the bytecode is refactored based on our proposed method. The refactored bytecode tend to give less test path than the original bytecode with about 75%. This extent of difference can be relative depending on the number of classes and functionalities of the code. The result also shows no difference in the application usage. As future work, automating the process of refactoring is in progress and we will also be looking at depicting the test path and displaying the difference in test case being generated.

ACKNOWLEDGEMENT

This research is supported under the Graduate Research Incentive Grants (GIPS), vote 1256, UniversitiTun Hussein Onn Malaysia.

REFERENCES

Ahmed, M., Ibrahim, R., and Ibrahim, N. Adaptation Model for Testing Android Application.In Proceedings of the 2015 International Conference on Computing Technology and Information Management.

Alves, E. L., Machado, P. D., Massoni, T., and Santos, S. T. 2013, May. A refactoring-based approach for test case selection and prioritization. In Proceedings of the 8th International Workshop on Automation of Software Test (pp. 93-99). IEEE Press.

Asaithambi, S. P. R., andJarzabek, S. 2013. Towards test case reuse: a study of redundancies in android platform test libraries. In Safe and Secure Software Reuse (pp. 49-64). Springer Berlin Heidelberg.

Grace, M. C., Zhou, Y., Wang, Z., and Jiang, X. 2012, February. Systematic Detection of Capability Leaks in Stock Android Smartphones. In NDSS.

Hecht, G., Rouvoy, R., Moha, N., andDuchien, L. 2015. Detecting Antipatterns in Android Apps (Doctoral dissertation, INRIA Lille).

International Data Corporation "Android and iOS Squeeze the Competition, Swelling to 96.3% of the Smartphone Operating System Market for Both 4Q14 and CY14, According to IDC" http://www.idc.com/getdoc.jsp?containerId=prUS25450615 (accessed 28/5/2015 Online)

Ko, M., Seo, Y. J., Min, B. K., Kuk, S., and Kim, H. S. 2012, May. Extending UML Meta-model for Android Application. In Computer and Information Science (ICIS), 2012 IEEE/ACIS 11th International Conference on (pp. 669-674). IEEE.

Kraemer, F. A. 2011. Engineering android applications based on UML activities. In Model Driven Engineering Languages and Systems (pp. 183-197).Springer Berlin Heidelberg.

Machado, P., Campos, J., and Abreu, R. 2013, August. MZoltar: automatic debugging of Android applications. In Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile (pp. 9-16).ACM.

Moghadam, I. H., andCinneide, M. O. 2012, March. Automated refactoring using design differencing. In Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on (pp. 43-52). IEEE.

Sony "ApkAnalyzer" http://developer.sonymobile.com/knowledge-base/tools/analyse-your-apks-with-apkanalyser/ (accessed 28/5/2015 Online).

Vekris, P., Jhala, R., Lerner, S., and Agarwal, Y. 2012, October. Towards verifying android apps for the absence of no-sleep energy bugs. In Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems (pp. 3-3). USENIX Association.

Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., and Wang, X. S. 2013, November. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security (pp. 1043-1054).ACM.

Zhang, Y., Huang, G., Liu, X., Zhang, W., Mei, H., and Yang, S. 2012, October. Refactoring android java code for on-demand computation offloading. In ACM SIGPLAN Notices (Vol. 47, No. 10, pp. 233-248). ACM.