Concolic Analysis for Android Applications

XXXXXXXXXXXXXXX
XXXXXXXXX
XXXXXXXXX
XXXXXXXXX
XXXXXXXXX, XX, XXX
XXXXXX@XXXXX.XXX

ABSTRACT

Mobile computing has become an integral part of our everyday lives, and the Android operating system has grown to be the most popular mobile platform. Unfortunately, Android applications are not immune to bugs, security vulnerabilities, and a wide range of other issues which are a common theme in all software. Concolic analysis is a powerful software testing process which has been applied to a variety of other areas including code clone detection and security-related activities.

We created a new, publicly available concolic analysis tool for analyzing Android applications: Concolic Analysis for Android (CAA), which performs concolic analysis on a raw Android application file (or source code) and provides concolic output in a useful and easy to understand format. The tool, detailed instructions, and source code are available on the project website: http://hiddenToKeepAnonymous. In the following work, we present the tool, references for installation and usage, and a brief outline for future research with the tool.

1. INTRODUCTION

Android has grown to become an extremely popular mobile platform with a wide variety of applications ('apps') varying in genre, function, and quality. As with all software, Android apps routinely suffer from bugs and security vulnerabilities. Static analysis tools can be extremely beneficial in assisting with these problems and can often quickly and accurately identify issues that developers would have otherwise missed [4, 11].

Concolic analysis is a powerful static analysis technique which has traditionally been used for software testing [8], security related activities [3], and code clone detection [6]. Traditional concolic tools such as JPF [10] and CATG[1] will not work on Android applications because they lack a main method that is typically required for concolic analysis tools. We are proposing a new tool, Concolic Analysis for Android (CAA), which allows users to perform concolic analysis on Android application (APK) source files with ease and without the need for a physical Android device or emulator. The tool not only includes the benefits of concolic static analysis, but provides concolic output which may be important for future work in clone detection and other comparison techniques [1, 6].

In the following work, we describe the need for our tool, provide details about the application and its design, and include basic usage instructions. The source code of CAA, installation instructions and further results may be found on our website: **http://hiddenToKeepAnonymous** .

2. RELATED WORK

There are a variety of concolic tools which have been created for Java and C based applications including JPF, CREST[2], CATG, and CUTE [8]. There are also several proposed techniques for applying concolic analysis to Android and mobile applications. Anand et al. [1] created ACTEve with a goal of alleviating the path explosion problem with concolic analysis. ACTEve is focused on event-driven applications and uses concolic analysis in order to generate feasible event-sequences for Android apps. Unfortunately this approach is often limited to short event sequences due to the resources required.

Similar to our tool, JPF-Android [9] verifies Android apps using JPF. A primary benefit of this technique is that it allows Android applications to be verified outside an emulator using JPF. While this work is profound, it differs from our tool in that it does not use concolic analysis to perform model checking and does not produce output about the functional nature of the app (as our tool does). Mirzaei et al. [7] described a process of testing Android applications through symbolic execution using custom Android libraries for JPF and simulated events through program analysis. While this work is substantial, it does not discuss the use of concolic analysis and does not appear to have been publicly released as a fully functional tool.

Blackshear et al. [2] created Droidel, a tool which summarizes the reflective behavior of Android apps using automatically generated application-specific stubs. Droidel creates a harness which can be used as a single entry point for an app, a similar problem which our tool was forced to overcome.

There are also many other powerful testing tools for Android apps[3]. Dynadroid[4] is a tool for creating inputs to unmodified Android apps. Google's own testing framework is also available to developers[5].

[1] https://github.com/ksen007/janala2

[2] https://code.google.com/p/crest/

[3] https://developer.android.com/tools/testing/testing-tools.html

[4] https://code.google.com/p/dynadroid/

[5] http://developer.android.com/tools/testing/testing_android.html

SAC'16, April 4-8, 2016, Pisa, Italy.

http://dx.doi.org/xx.xxxx/xxxxxxx.xxxxxxx ...$15.00.

3. CONCOLIC ANALYSIS

Concolic analysis uses the combination of concrete and symbolic values to analyze software and has been used for testing, identification of software clones, and the discovery of security vulnerabilities [3, 6, 8]. Concolic analysis was introduced by Sen et al. [8] in 2005, and has an advantage over symbolic analysis since its combination of concrete and symbolic values can be used to simplify constraints and precisely reason about complex data structures.

When an application is analyzed using concolic analysis, the execution path and symbolic constraints are stored in the *path condition*. An execution branch is then selected from this path condition which is provided to the constraint solver to be verified for legitimacy. If correct, concrete test inputs are then used to create a new achievable application path. If the new path is found to be unfeasible, another path is then selected. Concolic analysis attempts to traverse as many paths of the application as possible while limiting the path explosion problem through its use of concrete values [5]. Concolic analysis serves as the foundation of the CAA tool.

4. THE CAA TOOL

CAA uses several existing tools, which are:

- **Apktool**[6]: Decompiles APK files to standard Java .jar files.
- **Dex2Jar**[7]: A Java utility for decompiling Android applications. It extracts assets from the .jar file as well as decrypting the AndroidManifest.xml file.
- **Robolectric**[8]: A library designed to stub and mock out the Android runtime when testing a project outside of a standard runtime environment. In this project, it is used to grant access to code paths during concrete execution that otherwise would be unreachable without that framework.
- **JPF** [10]: Performs the actual concolic analysis. JPF is a good fit for this project since it is a freely available, widely used tool which provides the necessary extensibility options.

Several hurdles had to be overcome in the creation of our tool. First, the Android SDK does not support calls to arbitrary *main* functions, so it is therefore necessary to provide a wrapper for a decompiled Android APK file. This provides a single input to be used as the root node for the concolic parser's tree. Second, Android applications are not designed to be run outside an Android runtime environment, and the provided Android development libraries are insufficient as they are only stubs. This obstacle was overcome through the use of Robolectric, a dynamic Android mocking library which allows for greater coverage of Android code paths.

4.1 Overview of Architecture

The user first provides CAA a path to the APK file; CAA then executes a linear series of steps to perform concolic analysis on the target application. A high level overview of the process is described below and is shown in Figure 1.

1. Unpack APK resources and configuration files
2. Convert APK .dex files to Java .class files
3. Analyze user entry points into the application
4. Create a wrapper for decompiled APK
   (a) Create Java source files for wrapper from templates
   (b) Fill templates with entry point information and calls
   (c) Compile the wrapper

[6]https://code.google.com/p/android-apktool/
[7]https://code.google.com/p/dex2jar/
[8]http://robolectric.org/

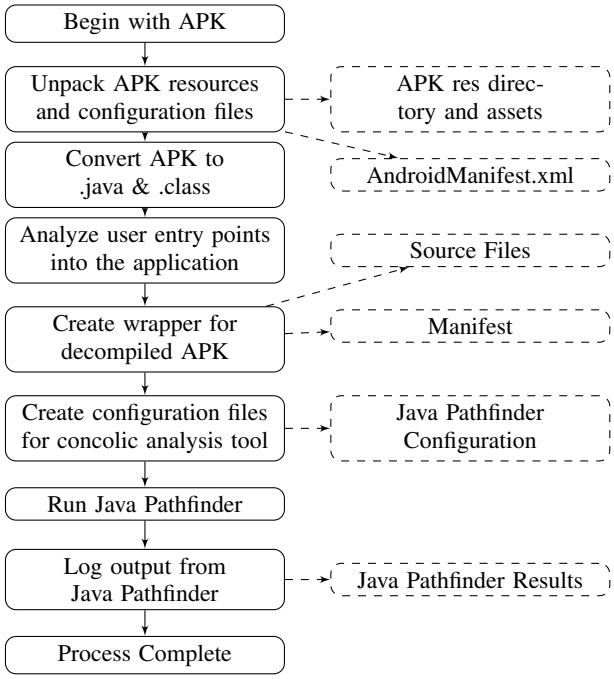

Figure 1: CAA Workflow

5. Create configuration files for concolic analysis tool
6. Run JPF (Java Pathfinder)
7. Log the output from JPF

The first step uses Apktool to produce the assets and configuration files which are later needed by Robolectric. All extracted files are placed in a special directory for later manipulation along with a copy of the targeted APK file.

The second step utilizes Dex2Jar to create the necessary Java .jar files from the APK file. The .jar format is required for later compilation and manipulation by the CAA tool. It exposes access to the internal code in a way that standard Java tools can easily work with.

The third step is to analyze the provided source from the generated jar file. Through the use of reflection, each class is dynamically loaded and analyzed for known inputs, such as an 'OnCreate' method of an activity. A blocklist is used to prevent excessive automated analysis of the Android libraries themselves, which are dynamically loaded to a custom classpath so that proper matching may occur. The types of discovered inputs are used to determine what functions need to be called and what kind of data they need to be sent by CAA and JPF.

The fourth and most complex step creates a custom wrapper jar against the created jar. Several template files are used to create raw Java source files with tokens. These tokens are replaced by a source writer in CAA, which interprets the analysis from the previous step. Calls to supported functions that the framework or user would trigger manually are automated in the source files.

In the final phases, a .jpf file is created to be used used by JPF to store arguments for passing to the concolic tool. This stores the targeted entry function provided by the wrapper jar, the functions that should have the analysis run on them, and the settings to enable concolic analysis. Finally, the output generated by the tool is saved for the user. This output may be useful to researchers and de-

velopers in a variety of ways including clone detection, uncovering defects, and analyzing the app's functional flow.

4.2 Usage Instructions

Once downloaded and installed using the instructions provided on our project website (**http://hiddenToKeepAnon**), the tool may be used with the following command: *"java -jar CAA-1.0.0.jar -apk $PATH_TO_APK"* where *$PATH_TO_APK* is the location of the APK file to be analyzed. A directory named "spawn" will be generated where several temporary artifacts of the process will be created. Results will be logged in a generated directory named "results" as text files similar in format to "*{$APKFILENAME}*.jpfout.txt." Thorough usage and installation instructions may be found on the project website. A sample of the instructions are shown in Figure 2 and an example status screen of the app is shown in Figure 3.

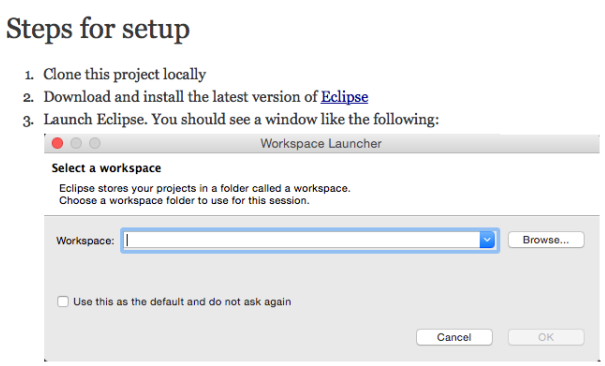

Figure 2: Example Usage Instructions from Website

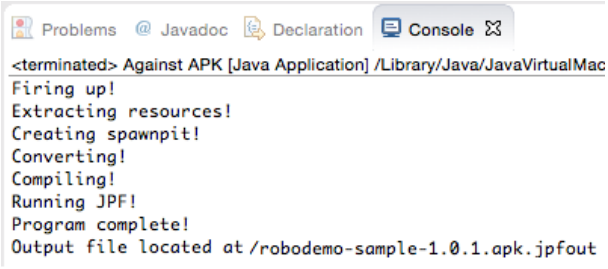

Figure 3: Example CAA Usage

5. LIMITATIONS & FUTURE WORK

While CAA represents a powerful and innovative static analysis tool, there are some notable limitations. A primary issue with the black box nature of this application is that certain Android apps require highly specific data at specific intervals, such as when communicating with servers. Robolectric has no way of knowing what an app expects back from specific calls, and thus cannot correctly mock it out; it can only mock out relatively simple or common Android API calls. This may cause certain code paths to be excluded from coverage if specific results for calls are expected.

Future work will be done to evaluate CAA against leading Android testing tools. Areas of comparison may include analysis time, amount of code coverage, and precision & recall of known errors. A few immediate uses of the tool include gaining a better understanding the functional nature of apps, seeking out redundant functionality, and finding defects.

6. CONCLUSION

We have presented CAA, a tool which analyzes Android applications using concolic analysis. We have made the tool, source code, and usage instructions available on our project website: **http://hiddenToKeepAnonymous**. We encourage others to use the tool not only for testing Android applications, but in their research as well.

7. REFERENCES

[1] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 59:1–59:11, New York, NY, USA, 2012. ACM.

[2] S. Blackshear, A. Gendreau, and B.-Y. E. Chang. Droidel: A general approach to android framework modeling. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2015, pages 19–25, New York, NY, USA, 2015. ACM.

[3] B. Chen, Q. Zeng, and W. Wang. Crashmaker: An improved binary concolic testing tool for vulnerability detection. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1257–1263, New York, NY, USA, 2014. ACM.

[4] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 576–587, New York, NY, USA, 2014. ACM.

[5] J. Jaffar, V. Murali, and J. A. Navas. Boosting concolic testing via interpolation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 48–58, New York, NY, USA, 2013. ACM.

[6] D. Krutz, S. Malachowsky, and E. Shihab. Examining the effectiveness of using concolic analysis to detect code clones. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, New York, NY, USA, 2015. ACM.

[7] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, Nov. 2012.

[8] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[9] H. van der Merwe, B. van der Merwe, and W. Visser. Execution and property specifications for jpf-android. *SIGSOFT Softw. Eng. Notes*, 39(1):1–5, Feb. 2014.

[10] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[11] M. S. Ware and C. J. Fox. Securing java code: Heuristics and an evaluation of static analysis tools. In *Proceedings of the 2008 Workshop on Static Analysis*, SAW '08, pages 12–21, New York, NY, USA, 2008. ACM.