2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)

CATE: Concolic Android Testing Using Java PathFinder for Android Applications

Patrick McAfee, Mohamed Wiem Mkaouer and Daniel E. Krutz
Department of Software Engineering
Rochester Institute of Technology
Rochester, NY, USA
Email: {pjm4439, mwmvse, dxkvse}@rit.edu}@rit.edu

***Abstract*—Like all software systems, Android applications are not immune to bugs, security vulnerabilities, and a wide range of other runtime errors. Concolic analysis, a hybrid software verification technique which performs symbolic execution along with a concrete execution path, has been used for a variety of purposes including software testing, code clone detection, and security-related activities. We created a new publicly available concolic analysis tool for analyzing Android applications: Concolic Android TEster (CATE). Building on Java Path Finder (JPF-SPF), this tool performs concolic analysis on a raw Android application file (or source code) and provides output in a useful and easy to understand format.**

I. Introduction

Concolic analysis is a powerful static analysis technique which has been traditionally used for software testing [4], security related activities [1], and code clone detection [3]. While there are a few concolic analysis tools for Java, none are immediately compatible with Android source code. Analysis tools such as Java PathFinder-Symbolic PathFinder (JPF-SPF)[1] and CATG[2] will not work on Android applications because the apps lack the main method which is typically required for concolic analysis tools. We are proposing a new tool, Concolic Android TEster[3] (CATE), 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. In addition to providing the benefits of concolic static analysis, its generated concolic output may be important for future work in clone detection and concolic analysis based techniques [2]. Similar to our tool, JPF-Android[4] 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. This tool differs from CATE 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).

II. Concolic Android Analysis

There were several hurdles we had to overcome when creating 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, and the provided Android development libraries are insufficient as they are only stubs. This obstacle was overcome through the use of Robolectric[5], a dynamic Android mocking library which allows for greater coverage of Android code paths. CATE executes a linear series of steps to provide the results of concolic analysis. An overview of the process is shown in Figure 1.

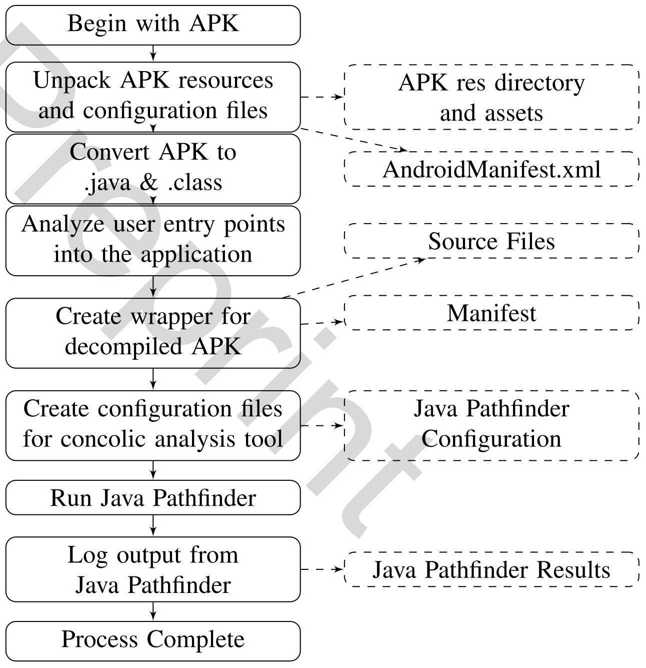
Begin with APK
Unpack APK resources and configuration files
APK res directory and assets
AndroidManifest.xml
Convert APK to .java & .class
Analyze user entry points into the application
Source Files
Create wrapper for decompiled APK
Manifest
Create configuration files for concolic analysis tool
Java Pathfinder Configuration
Run Java Pathfinder
Log output from Java Pathfinder
Java Pathfinder Results
Process Complete

Fig. 1. CATE Workflow

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

[1]http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc
[2]https://github.com/ksen007/janala2
[3]http://www.se.rit.edu/~dkrutz/cate/
[4]https://bitbucket.org/heila/jpf-android/src
[5]http://robolectric.org/

The second step utilizes dex2jar[6] to produce the necessary Java .jar files from the APK file. The .jar format is required for later compilation and manipulation by the CATE tool. It exposes access to the internal code in a way that standard Java tools can easily work with. It also provides readable source code, an invaluable resource during development.

The third step is analyzing the source code 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 class-path so that proper matching can happen. The types of inputs found are used to determine what functions need to be called and what kind of data they need to be sent by CATE and JPF-SPF.

The fourth and most complex step creates a custom wrapper jar against the created jar. Template files are used to create raw Java source files with tokens. These tokens are replaced by a source writer in CATE, which interprets the analysis from the previous step. Then, calls to supported functions that the framework or user would trigger manually are automated in the source files. There are two .java files and a manifest file created from this process, as well as a .jpf file. The first Java file is a wrapper that makes all of the aforementioned calls to the jar converted from the APK file and wraps those calls in a single function as a JUnit test. Robolectric, the Android mocking library being used, operates as a JUnit TestRunner and thus the wrapper function must be a test to utilize the mocks. The second Java file is the wrapper runner whose purpose is loading the wrapper's tests into JUnit and firing them from inside a single entry point. This entry point is then exposed to JPF-SPF and indirectly provides access to the underlying functions from the APK file. The final file is a custom manifest that references all dependencies as well as the jar converted from the APK. The newly created Java source and manifest are packaged into a custom wrapper jar to be used in the next step of the process.

```
8    checkcast
11   putfield java.util.HashMap.table
14   aload_0
15   iconst_0
16   putfield java.util.HashMap.hashSeed
19   aload_0
20   aconst_null
21   putfield java.util.HashMap.entrySet
24   iload_1
```

Listing 1. Example Concolic Output

The fifth and sixth steps build on part of the fourth. During the creation of files from templates, a .jpf file is created. This .jpf file is used by JPF-SPF to store arguments to pass to the concolic tool. In particular, 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. A small example of this output is shown in Listing 1, and more complete results may be found on the project website as well as instructions on how to install and use CATE. This output may be useful to researchers and developers in a variety of ways including clone detection, uncovering defects and analyzing the app's functional flow.

[6]https://sourceforge.net/projects/dex2jar/

III. Analysis and future work

The most prominent area that concolic analysis has been applied to thus far is software testing, specifically for dynamic test input generation, test case generation, and bug detection. These are areas which not only affect conventional software, but Android apps as well. While there have been many testing tools created to assist in ensuring high quality apps, we believe that our tool can assist in detecting a variety of issues which were previously not discoverable using a static analysis based approach for Android apps. Our tool may provide valuable assistance in this area for both developers and researchers in a variety of these problematic areas.

While CATE represents a powerful and innovative static analysis tool, there are some notable limitations. The targeted coverage is limited to the activity lifecycle startup and is also restricted by the intentional black box testing nature of usage with mocks. The primary issue with the black box nature is that certain Android apps require highly specific data at certain 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 more simple or common Android API calls. This may cause certain code paths to be excluded from coverage if specific results for calls are expected.

Previous research has found that 86% of malware samples were repackaged version of legitimate apps [5]. Since concolic analysis has been used in clone detection, it can likely be a valuable asset for detecting repackaged apps since they are essentially just clones or copies of legitimate apps or libraries. Furthermore, CATE's analysis with other existing Android testing tools such as JPF-Android and EvoDroid[7].

References

[1] 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.

[2] 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.

[3] D. Krutz and E. Shihab. Cccd: Concolic code clone detection. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 489–490, Oct 2013.

[4] 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.

[5] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.

[7]http://www.sdalab.com/projects/evodroid