Using JPF to Perform Concolic Analysis on Android Applications

Patrick McAfee, Nuthan Munaiah, Samuel A. Malachowsky and Daniel E. Krutz
Software Engineering Department
Rochester Institute of Technology
1 Lomb Memorial Drive
Rochester, NY, USA
{pjm4439, nm6061, samvse, dxkvse}@rit.edu

ABSTRACT

Mobile computing has become an important 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, a common theme in all software. Concolic analysis, a hybrid software verification technique which performs symbolic execution along 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 Analysis for Android (CAA). Building on Java Path Finder (JPF), 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. Included in this paper are an introduction to the root concepts, a description of the tech stack used within the tool, and basic usage instructions. The tool, detailed instructions, and source code is available on the project website: http://www.se.rit.edu/~dkrutz/caa/. In the following work, we present the tool, references for installation and usage, and a brief outline for future research with the tool.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – Testing tools

Keywords

Concolic Analysis, Java PathFinder, Testing, Verification

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 issues and can often quickly and accurately identify issues that developers would have otherwise missed [4, 17].

Concolic analysis is a powerful static analysis technique which has been traditionally used for software testing [13], security related activities [3], and code clone detection [7, 8]. While there are a few concolic analysis tools for Java, none are immediately compatible with Android source code. Traditional analysis tools such as JPF [16] and CATG[1] will not work on Android applications because the apps lack a main method which is typically required for concolic analysis tools. We are proposing a new tool, Concolic Analysis on 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, 7, 8].

The CAA tool executes seven primary steps: (1) Unpacking the Android application, (2) Conversion of APK into a .jar file, (3) Analysis of entry points into the application, (4) Creation of a wrapper for the decompiled APK, (5) Creation of configuration files for concolic analysis tool, (6) Running JPF, and (7) Logging output from JPF. 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[2].

This paper is organized as follows: Section 2 describes related work and Section 3 provides a background on concolic analysis. Section 4 describes the tool we created and the tool, limitations and future work are discussed in Section 5. Our work is concluded in Section 6.

2. RELATED WORK

Concolic analysis is just one type of static analysis which has been performed on software ranging from large enterprise systems, to mobile software. There have been a diverse array of static analysis tools created for Android apps for a variety of reasons ranging from boosting an app's performance [2] to the discovery of privacy leaks [11]. There are an assortment of concolic tools which have been created for Java and C based applications. Some include JPF, CREST[3], CATG, and CUTE [13]. There are also several proposed techniques for applying concolic analysis to Android and mobile applications. Anand et al. [1] created a method known as ACTEve with a goal of alleviating the path-explosion problem with concolic analysis. ACTEve is focused on event-driven applications such as

[1]https://github.com/ksen007/janala2
[2]http://www.se.rit.edu/~dkrutz/caa/
[3]https://code.google.com/p/crest/

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

smartphone 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 [15] 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. [12] 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.

While none use concolic analysis, there are other powerful testing tools for Android apps. Dynadroid[4] is a tool for creating inputs to unmodified Android apps. EvoDroid [9] is a tool which combines Android specific program analysis and an evolutionary algorithm in order to achieve higher levels of code coverage in apps. Google's own testing framework is also available to developers[5].

3. CONCOLIC ANALYSIS

Concolic analysis uses the combination of concrete and symbolic values to analyze software and has been used for testing: assisting with unit test creation, identification of software clones, and the discovery of security vulnerabilities [5, 7, 8, 13]. Concolic analysis was introduced by Sen et al. [13] in 2005, and has an advantage over symbolic analysis since the combination of concrete and symbolic values can be used to simplify constraints and precisely reason about complex data structures [10].

When an application is executed using concolic analysis, the execution path along with 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. However, if the new path is found to be unachievable, another application path is then selected. Using this process, 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 [6].

```
1  void f(int a, int b){
2     int c = 2*b;
3     if(a=100000){
4        if(a<c){
5           assert(0); //error
6        }
7     }
8  }
```

Listing 1: Code to be examined by Concolic Analysis

As an example, Listing 1 displays a function which is to have concolic analysis performed upon it, and Figure 1 shows its data flow. The analysis process would first begin with an arbitrary value being assigned to a and b. For the concrete execution, a=b=1. Line #2 would set c to be 2, and the *if* statement in the 3rd line will fail since a $\neq$ 100000. The symbolic execution will follow the same path taken by the concrete execution, but will merely treat a and b as symbolic variables. C will be set to the expression 2b and will make note that a $\neq$ 100000 since the test in line 3 failed. This is known as a path condition and will need to be true for every execution following this same path. The goal is to examine every possible path of the application.

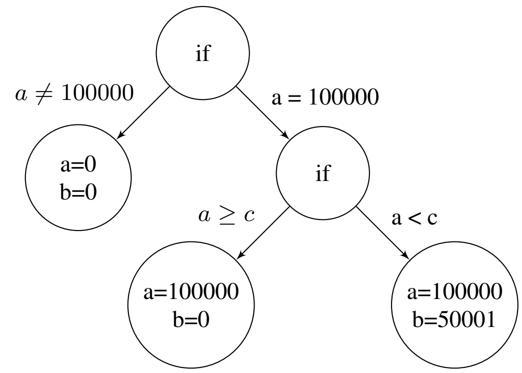

Figure 1: Concolic Analysis Flow

Concolic analysis serves as the foundation of the CAA tool, which is described in the next section.

4. CONCOLIC ANALYSIS FOR ANDROID (CAA) TOOL

In the following sections, we will describe the main components which comprise the tool, an overview of the tool, and then conclude with installation and usage instructions.

4.1 Main Components

The core of CAA is based on linking several previously existing tools together in order to provide a framework where concolic analysis can be run with a single process. CAA is designed as an automated toolchain, utilizing other disparate tools to do the more complex tasks. They are as follows:

- **Apktool**[6]: A utility for decompiling APK files into standard Java .jar files.
- **Dex2Jar**[7]: A Java utility for decompiling Android applications. This tool extracts the UI resources and other assets from the .jar file, as well as decrypting the AndroidManifest.xml file.
- **Robolectric**[8]: A library specially designed to stub and mock out the Android runtime when testing a project outside of a standard runtime environment. In the context of this project, it is used to grant access to code paths during concrete execution that other would be unreachable without that framework.
- **JPF** [16]: Performs the actual concolic analysis. While there are other tools that provide this functionality against vanilla .jar files, such as jCUTE[9] and CATG, JPF is a best fit for this project, particularly due to its Symbolic optional module which includes the actual concolic analysis. Additionally, it is actively supported with a robust development community (including NASA), which was critical to this project's development. One feature that other tools did not include was the ability to configure a flexible classpath at runtime with

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

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

[6]https://code.google.com/p/android-apktool/

[7]https://code.google.com/p/dex2jar/

[8]http://robolectric.org/

[9]http://osl.cs.illinois.edu/software/jcute/

multiple directories. This functionality was required for the dynamic compilation described in the design portion of this work.

There were several hurdles we had to 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, 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.

4.2 Overview of Architecture

CAA accepts a path to an APK and executes a linear series of steps to provide the results of concolic analysis. A high level overview of the process is described below and is shown in Figure 2.

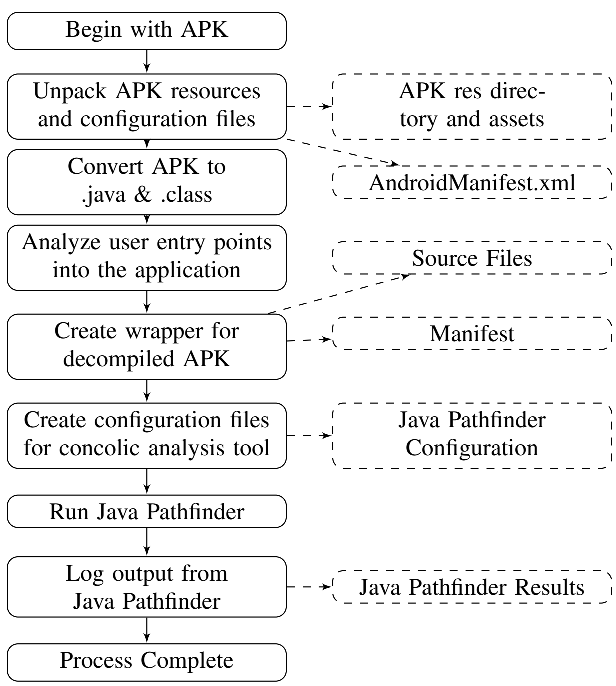

Figure 2: CAA Workflow

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 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
5. Create configuration files for concolic analysis tool
6. Run JPF
7. Log output from JPF

The first step of the process 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. All subsequent modification is done in this directory.

The second step utilizes Dex2Jar to produce 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. It can also provide readable source code, an invaluable resource during development. This jar is also stored within the spawn directory.

The third step is analyzing 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 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 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. 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 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.

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 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 2, and more complete results may be found on the project website. 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.

```
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 2: Example Concolic Output

4.3 Usage Instructions

Once downloaded and installed using the instructions provided on our project website (**http://www.se.rit.edu/~dkrutz/caa/**), 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 path to the APK file to be analyzed. A directory named "spawn" will be created where several temporary artifacts of the process will be created. Results will be logged in a created directory named "results" as text files similar in format to "*{$APK FILENAME}*.jpfout.txt." The results directory is located in the same directory as the application. A thorough example set of instructions on running the tool may be found on the project website. A sample of the setup instructions are shown in Figure 3 and an example status screen of the app is shown in Figure 4. A portion of the concolic output screen is shown in Figure 5.

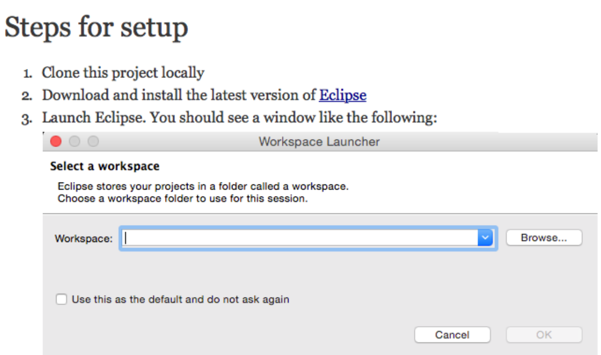

Figure 3: Example Usage Instructions from Website

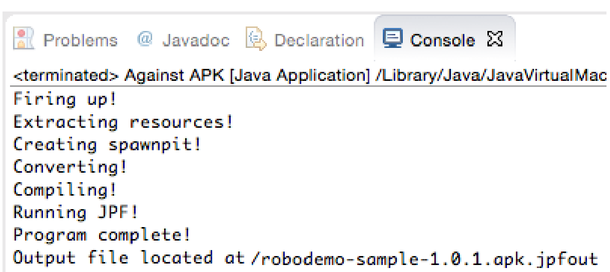

Figure 4: Example CAA Usage

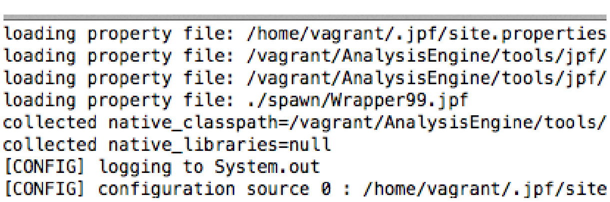

Figure 5: Example Portion of Tool Output

5. DISCUSSION

In the following sections we will discussion limitations of the tool, future improvements which can be made to it, and the areas that it can be applied to for both research and developmental purposes.

5.1 Limitations

While CAA 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 of this application 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.

Available APKs are currently theoretically compatible with both and are convertible by Dex2Jar. In the future, this may not be the case as incompatibilities are discovered and the Dalvik runtime becomes antiquated or is no longer supported. Finally, CAA is limited by the tools it relies on and the idiosyncrasies and issues associated with them. As an example, the development of CAA revealed several issues and bugs in JPF's implementation of reflection, some of which are still outstanding and have the potential to affect the reliability of CAA.

5.2 Applications of Tool

Concolic analysis provides several benefits, which this tool is also capable of providing. Capability or Permission Re-Delegation occurs when a malicious app uses the permissions of a vulnerable app. For example a benign app could be granted the `CALL_PHONE` permission for legitimate reasons by the user. A malicious app could used an exposed intent to use this permission for harmful reasons. There have been several proposed techniques for both avoiding and detecting capability leaks including fuzzing [19].

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 [13, 18]. 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 assisting in detecting a variety of issues which were previously undiscoverable using a static analysis based approach for Android apps. We believe that our tool could provide valuable assistance in this area for both developers and researchers in a variety of these problematic areas.

5.3 Future Work

There are many improvements that can be made to CAA. As an example, CAA is limited to inefficiently processing one APK file at a time; the concurrent processing of multiple applications would make the usage of the tool more efficient, allowing the application to be more easily utilized by other tools. This change could, for example, allow a user to compare two similar apps and run heuristics on the generated output more quickly (a use case that led to the creation of CAA).

The source writer could also be expanded to provide more coverage. For example, a filter for Android services could be added and the wrapper files could be compiled to target the launch and processing of these application parts. This could then be expanded upon as the Android SDK grows and changes, allowing new entry points and data sources to be covered.

No significant amount of work has been done to compare CAA to other existing Android testing tools. Areas of comparison could include analysis time, amount of code coverage, and precision & recall of known errors. A few tools which CAA should be compared to include: ACTEve [1], JPF-Android [15] and EvoDroid [9].

Previous research has found that 86% of malware samples were repackaged version of legitimate apps. [20]. Repackaged apps are first reverse engineered from legitimate apps, and malicious code is then injected into the app's source code. The app is then recompiled into an apk and is redistributed as the legitimate version of the app [14]. Concolic has been used to detect code clones [7, 8], which are functionality equivalent portions of code, which may be syntactically different. Future work can be done to apply CAA in the detection of these fraudulent apps, since repackaged apps are essentially just clones or copies of legitimate apps or libraries.

6. CONCLUSION

We have presented CAA, a tool that analyzes Android applications using concolic analysis. While there are numerous other testing tools for Android applications (and even some which use concolic analysis), this is the first known freely available tool of its kind which is able to perform concolic analysis using only the source code of the Android application.

Through a series of seven steps, CAA extracts the source code of the application (using existing tools), dynamically loads the extracted Java class files looking for known inputs which are used in the customer wrapper, then generates the concolic analysis output, which may be used in a variety ways.

We have made the tool, source code, and usage instructions available on our project website: **http://www.se.rit.edu/~dkrutz/caa/**. 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. Anwer, A. Aggarwal, R. Purandare, and V. Naik. Chiromancer: A tool for boosting android application performance. In *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*, MOBILESoft 2014, pages 62–65, New York, NY, USA, 2014. 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] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing.

[6] 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.

[7] 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.

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

[9] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 599–609, New York, NY, USA, 2014. ACM.

[10] R. Majumdar and K. Sen. Hybrid concolic testing. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 416–426, May 2007.

[11] S. Matsumoto and K. Sakurai. A proposal for the privacy leakage verification tool for android application developers. In *Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication*, ICUIMC '13, pages 54:1–54:8, New York, NY, USA, 2013. ACM.

[12] 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.

[13] 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.

[14] C. Soh, H. B. K. Tan, Y. L. Arnatovich, and L. Wang. Detecting clones in android applications through analyzing user interfaces. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC '15, pages 163–173, Piscataway, NJ, USA, 2015. IEEE Press.

[15] 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.

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

[17] 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.

[18] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 249–260, New York, NY, USA, 2008. ACM.

[19] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan. Intentfuzzer: Detecting capability leaks of android applications. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 531–536, New York, NY, USA, 2014. ACM.

[20] 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.