A Framework for Automated Security Testing of Android Applications on the Cloud

Sam Malek, Naeem Esfahani, Thabet Kacem, Riyadh Mahmood, Nariman Mirzaei, Angelos Stavrou
Computer Science Department
George Mason University
{smalek, nesfaha2, tkacem, rmahmoo2, nmirzaei, astavrou}@gmu.edu

I. INTRODUCTION

App markets are stirring a paradigm shift in the way software is provisioned to the end users. The benefits of this model are plenty, including the ability to rapidly and effectively acquire, introduce, maintain, and enhance software used by the consumers. By providing a medium for reaching a large consumer market at a nominal cost, app markets have leveled the playing field, allowing small entrepreneurs to compete with the largest software development companies of our times. The result of this has been an explosive growth in the number of new apps for platforms, such as Mac, Android, and iPhone, that have embraced this model of providing their consumers with diverse, up-to-date, and low cost apps.

This paradigm shift, however, has given rise to a new set of security challenges. In parallel with the emergence of app markets, we have witnessed increased security threats that are exploiting this model of provisioning software. Arguably, this is nowhere more evident than in the Android market, where numerous cases of apps infected with malwares and spywares have been reported [4]. There are numerous culprits here, and some are not even technical, such as the general lack of an overseeing authority in the case of open markets and inconsequential implication to those caught providing applications with vulnerabilities or malicious capabilities.

However, from a technical standpoint, the key obstacle is the ability to rapidly assess the security and robustness of applications submitted to the market. The problem is that security testing is generally a manual, expensive, and cumbersome process. This is precisely the challenge that we have begun to address in a DARPA (Defense Advanced Research Projects Agency) sponsored project targeted at the development of a framework that aids the analysts in testing the security of Android apps. The framework is comprised of a tool-suite that given an application automatically generates and executes numerous test cases, and provides a report of uncovered security vulnerabilities to the human analyst. We have focused our research on Android as (1) it provides one of the most widely used and at the same time vulnerable app markets, (2) it dominates the smartphone consumer market, and (3) it is open-source, lending itself naturally for research and experimentation in the laboratory.

Security testing is known to be a notoriously difficult activity. This is partly because unlike functional testing that aims to show a software system complies with its specification, security testing is a form of *negative testing*, i.e., showing that a certain behavior does not exist in the system.

A form of automated security testing that does not require test case specification or significant upfront effort is *fuzz testing,* or simply *fuzzing* [6]. In short, fuzzing is a form of negative software testing that feeds malformed and unexpected input data to a program with the objective of revealing security vulnerabilities. Programs that are used to create and examine fuzz tests are called *fuzzers*. In the past, fuzzers have been employed by the hacking community as one of the predominant ways of breaking into a system and they have been very successful at it [6]. An SMS protocol fuzzer [5] was recently shown to be highly effective in finding severe security vulnerabilities in all three major smartphone platforms, namely Android, iPhone, and Windows Mobile. In the case of Android, fuzzing found a security vulnerability triggered by simply receiving a particular type of SMS message, which not only kills the phone's telephony process, but also kicks the target device off the network [5].

In spite of the success stories, there is a lack of sophisticated frameworks for fuzz testing apps, in particular those targeted at smartphone platforms, including Android. There are a few available fuzzers, such as Android's Monkey [1], that generate purely random test cases, and thus often not very effective in practice. Moreover, fuzz testing is generally considered to be a time consuming and computationally expensive process, as the space of possible inputs to any real-world program is often unbounded.

We are addressing these shortcomings by developing a scalable approach for intelligent fuzz testing of Android applications. The framework *scales* both in terms of code size and number of applications by leveraging the unprecedented computational power of cloud computing. The framework uses numerous heuristics and software analysis techniques to *intelligently* guide the generation of test cases aiming to boost the likelihood of discovering vulnerabilities. Our research aims to answer the following overarching question: *Given advanced software testing techniques and ample processing power, what software security vulnerabilities could be uncovered automatically?* The framework enables the fledgling app market community to harness the immense computational power at our disposal together with novel automated testing techniques to quickly, accurately, and cheaply find security vulnerabilities.

In the next section, we provide an overview of this framework and its underlying architecture.

II. Framework Overview

Figure 1 shows an overview of the framework. As depicted, parts of the framework execute on a cloud platform to allow for the generation and execution of large number of test cases on many instances of a given application.

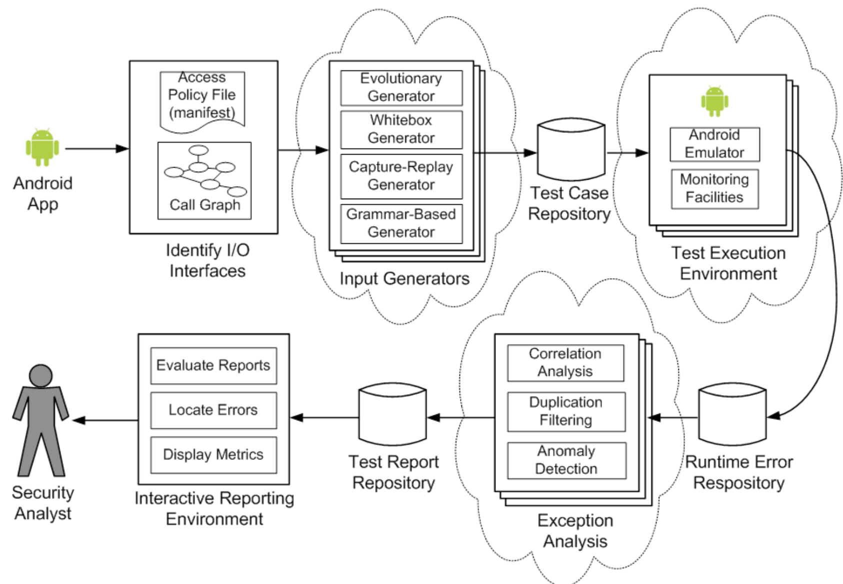

Figure 1. Overview of the framework. Components contained in the bubble indicate the parts that can execute in parallel on the cloud.

Given an Android application for testing, the first step is to automatically *Identify Input/Output Interfaces*, as shown in the top left corner of Figure 1. An application's input interfaces represent the different ways in which it can be invoked by either the execution environment or user. They may include GUIs, network interfaces, files, APIs, messages, etc. An application's output interfaces are also important, as their abnormal behavior could lead to detecting vulnerabilities. We are leveraging a variety of analysis techniques to identify an application's interfaces, even those that may be hidden or disguised. For instance, we have developed a program analysis technique to identify all the graphical user interfaces widgets through which the user can interact with a system. The program analysis technique leverages numerous sources of information obtained from the app's implementation, including the app's call graph, abstract syntax tree, and manifest file that provides lots of meta-information about the application's architecture and it access permissions. Here, if the source code of an Android app is not available, we reverse engineer its APK file, which is the installation package file, using one of the existing tools (e.g., dextojar [2]).

Following that, and as shown in Figure 1, *Input Generator* engines are leveraged to create the candidate test cases. We are developing several different types of input generators, each of which would leverage a different set of heuristics for guiding the generation of test cases. This allows for diversity among the test cases, as each input generator provides unique strengths, enabling the framework to achieve good coverage and test a wide-range of boundary conditions. Since some of the generators are computationally expensive and may take a significant amount of time to run, the framework executes many instances of them in parallel on the cloud. For instance, we are revising *Java Pathfinder*—a Java symbolic execution engine previously developed at NASA Ames—to be able to generate test cases for Android apps. Using the Android-specific Java Pathfinder, we are able to systematically execute an Android app to generate test cases that exercise different parts of the app, and thus achieve good code coverage.

Following the generation of test cases, the *Test Execution Environment* is deployed to simultaneously execute the tests on numerous instances of the same application. We execute the majority of the test cases on virtual nodes running the *Android Emulator* on the cloud. However, a cluster of actual Android devices is also employed for executing a small subset of the tests that require high fidelity. Several Android-specific *Monitoring Facilities* (e.g., *Intent Sniffer* [3]) are leveraged and deployed to collect runtime data as tests execute. The monitoring facilities record issues and errors (e.g., crashes, exceptions, access violations, resource thrashing) that arise during the testing in the *Runtime Error Repository*.

Exception Analysis engine (shown in Figure 1) then investigates the *Runtime Error Repository* to correlate the executed tests cases to the reported issues, and thus potential security vulnerabilities. Moreover, the *Exception Analysis* engine prunes the collected data to filter any redundancy, since the same vulnerability may be encountered by multiple test cases. It also looks for anomalous behavior, such as performance degradations, which may also indicate vulnerabilities (e.g., an input test that could be used to instigate a denial of service attack). The results of these analyses are stored in a *Test Report Repository*, which is then used by the *Interactive Reporting Environment* to enable the security analyst to evaluate the application's robustness and understand its vulnerabilities.

Acknowledgment

This research is supported by grant N11AP20025 from Defense Advanced Research Projects Agency.

References

[1] Android Monkey. http://developer.android.com/guide/developing/tools/monkey.html

[2] Dextojar. from http://code.google.com/p/dex2jar/

[3] Intent Sniffer. http://www.isecpartners.com/mobile-security-tools/

[4] Malicious Mobile Threats Report 2010/2011, White paper, Juniper Networks Global Threat Center Research. http://www.juniper.net/us/en/local/pdf/whitepapers/2000415-en.pdf

[5] C. Mulliner, and C. Miller. Fuzzing the Phone in your Phone. *Black Hat*, USA, July 2009.

[6] A. Takanen, et al. Fuzzing for Software Security Testing and Quality Assurance. *Artech House, Information Security and Privacy Series*, Norwood, MA, 2008.