Measuring the Effects of a Record and Replay Tool on Android Application Fuzz Testing

Richard Lee[1,2], Cyril Saade[2], Dr. Matthias Büchler[2], Dr. Manuel Egele[2]

Manhasset High School, 200 Memorial Place, Manhasset, NY 11030[1]

Electrical and Computer Engineering Lab, Boston University, 8 St. Mary's Street, Boston, MA 02215[2]

Abstract

Fuzz testing (fuzzing) is a security testing approach where an automated and randomized input is sent to an application in order to identify vulnerabilities. Since the input is randomized, fuzzing is mainly focused on application crashes. Letting an application crash is a good sign that there is a vulnerability that can be exploited. However, the randomized nature of fuzzing implies that certain parts of applications, such as login or payment screens, cannot be exercised, meaning that these parts may rarely be fuzz tested by developers. One solution to this problem is to use a record and replay tool for user actions: a record and replay tool could be used to aid fuzzing on parts of an application that require specific user interaction, which would enable fuzz testing on functionalities that normally may not be accessible to standard fuzzing. The purpose of this project is to measure the increase in code coverage when fuzzing is combined with a record and replay tool. In this study, we use RERAN, a record and replay tool published by Gomez et al. in 2013, to change the state of an application by replaying actions, before using the MonkeyRunner tool to do fuzz testing. We use Android's Java Platform Debugger Architecture to measure code coverage and compare how fuzz testing on apps with and without RERAN changes the amount of code executed. We measure code coverage using a Python script that sets 'hooks' within an application and logs when a hook is passed in the program execution.

Introduction

Fuzzing is a security testing approach where automated and randomized input is sent to an application to test how it will react. If application crashes occur during fuzz testing, it indicates that there are vulnerabilities in implementation. However, the random nature of fuzz testing implies that it may rarely reach and fuzz test certain parts of an app, because login screens or payment requests require some sort of password to advance to the next screen. This means that developers would have to manually enter a password and/or username every time they wanted to fuzz test parts of the application behind such functions, drastically reducing productivity. One way to mitigate these issues is to use a record and replay tool: the tool could automate logins and or other specific functions in an application so that fuzzing can be initiated from a different part of the application.

In this study, we conduct fuzz testing on applications that run on the mobile operating system Android. The Android Debug Bridge (adb) includes the Monkey tool, which sends randomized input to an app. Although monkey was designed for stress testing apps, in this experiment we use it to fuzz applications. For a record and replay tool, we use RERAN, which is a: "timing and touch-sensitive record and replay for Android"[1] that allows for user input on an Android to be recorded and executed to replay the recorded actions. By pairing RERAN with Monkey, fuzz testing can occur on more obscure sections of an application. We measure how RERAN improves fuzzing by measuring code coverage. Code coverage is defined as the percentage of code in the application that is executed. The more code coverage a fuzz test has, the more complete the test is. RERAN should significantly increase code coverage because it gives monkey access to the rest of the application behind the login screen. The purpose of this project is to measure the increase in code coverage when we combine fuzzing with the record and replay tool RERAN.

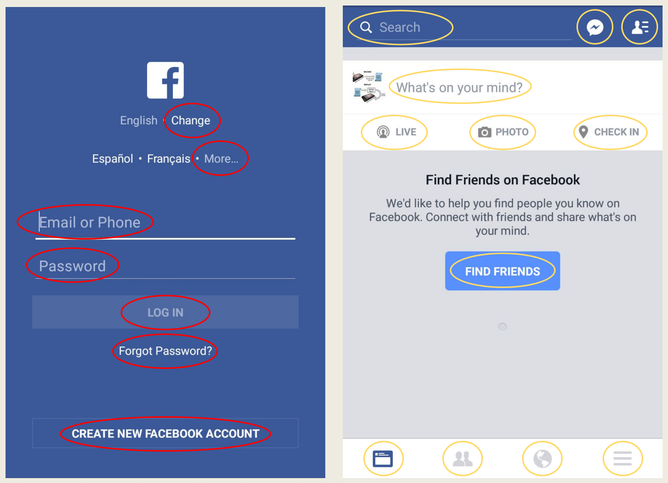


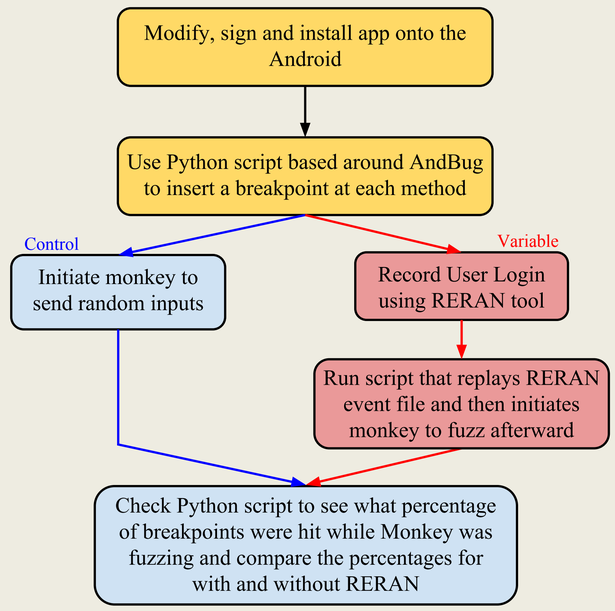
Figure 1. Facebook, a social media app, includes a login screen with only 7 potential inputs for fuzzing. However, once logged in, there is a multitude of other input boxes, with 12 potential inputs on just the home screen. By automating the login process with a record and replay tool, fuzzing can be more easily conducted on the rest of the app behind the login screen.

Methodology

The control in this experiment is to test fuzzing unaided by any other tools. The variable in this study is to test fuzzing aided by a record and replay tool that executes prior to any fuzz testing and moves the app to another state.

We used the Java Platform Debugger Architecture in conjunction with AndBug[2], an Android Debugging Library written in Python. AndBug has the capability of inserting a breakpoint in different parts of an application given just the class and method name. By inserting a breakpoint at each method in a program, we can effectively test for code coverage because we can find how many methods in the program were executed, and therefore know how much code was run. In this experiment, a breakpoint does not need to halt the program; instead, it only needs to log that it was hit, and let the program continue.

Below is a visual representation of the methodology of this experiment:



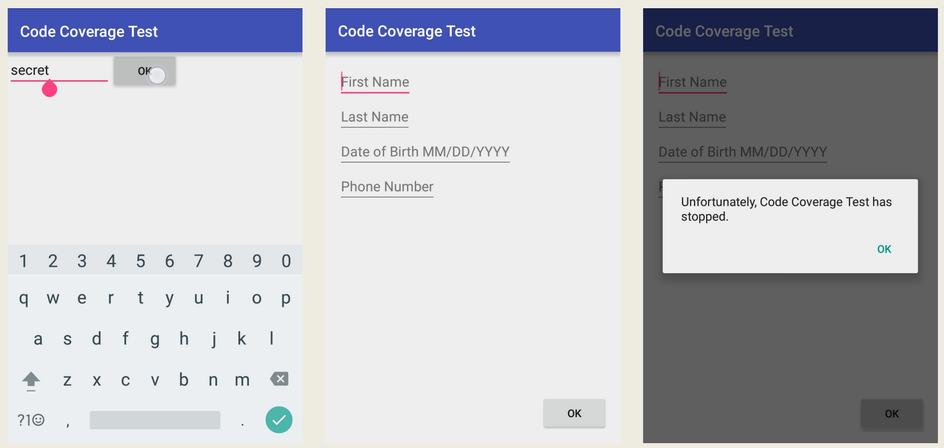


Figure 2. We designed a small Android application with only a few classes and methods to test whether fuzzing would increase the code coverage. The first screen of the application contained an input field and button that would wait for the string, "secret" to be entered and the button pushed, before allowing the app to advance. This was in basic simulation of a login screen, where specific input would be required and fuzzing would not be able to pass it. The second screen contained a 'bug' where there were several input fields and a button. If the button was pushed, then the app would crash, simulating an input field that has an error in implementation.

Results

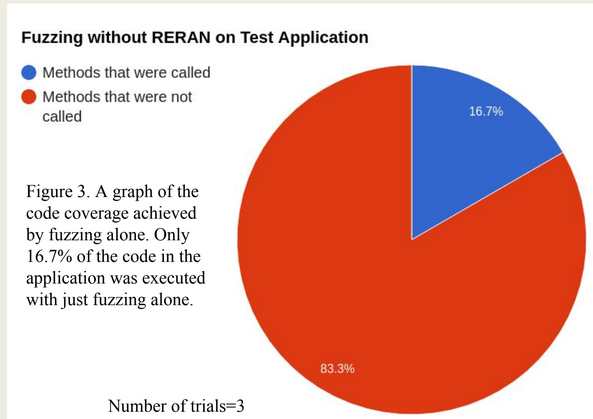


Figure 3. A graph of the code coverage achieved by fuzzing alone. Only 16.7% of the code in the application was executed with just fuzzing alone.

Number of trials=3

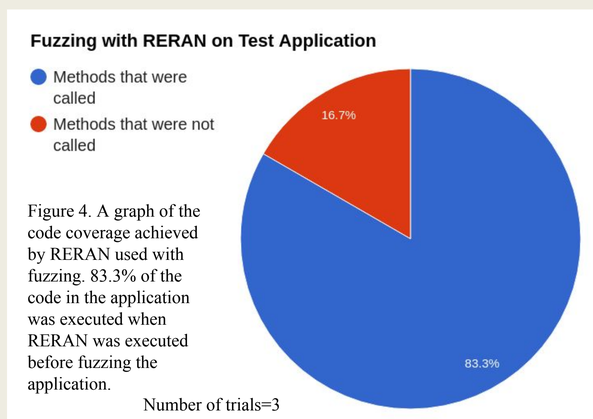


Figure 4. A graph of the code coverage achieved by RERAN used with fuzzing. 83.3% of the code in the application was executed when RERAN was executed before fuzzing the application.

Number of trials=3

Discussion

- Combining a record and replay tool with fuzzing significantly increases code coverage; in the current test case, from 16.7% to 83.3%.

- Developers that wish to fuzz test their applications could use a record and replay tool to record navigation to each page in their application, and run the recorded file right before they would like to test that specific part of the application

- The application contained a "bug" where the application would crash if the 'OK' button on the second screen was pressed. Fuzzing without the replay tool was unable to find this flaw, while fuzzing with RERAN was able to find the flaw repeatedly.

- Limitations: More complex applications could not be tested; each method in an application requires a breakpoint, and each breakpoint requires a certain amount of memory, so more complex apps consumed all of the Android's available memory, preventing us from fuzz testing them.

References

[1]Gomez, Lorenzo; Neamtiu, Iulian; Azim, Tanzirul; Millstein, Todd. RERAN: Timing- and Touch-Sensitive Record and Replay for Android. International Conference on Software Engineering (ICSE). 2013. 35th.

[2]Dunlop, Scott. AndBug -- A Scriptable Android Debugger. GitHub. https://github.com/swdunlop/AndBug

Acknowledgements

We would like to acknowledge Dr. Ayse Coskun for her valuable input on how to use the Android Operating System and how to better integrate RERAN with fuzz testing.

Conclusions

- By combining record and replay tools with fuzz testing, developers have the capability to find flaws or bugs in application implementation that might otherwise not have been found with fuzzing alone.

- To test larger scale applications, a special version of the Android Runtime (ART) would need to be designed; ART manages how the Android Operating System handles breakpoints, but currently it does not allow for thousands of breakpoints to be placed. ART iterates through the entire list of breakpoints every time it encounters a breakpoint, meaning that when there are thousands of breakpoints, the entire application freezes, waiting for ART to finish iterating through the list of breakpoints. A method of resolving this would be to design a version of the Runtime that manages breakpoints more efficiently, so the application does not freeze when the Runtime hits a breakpoint.