BLACK-BOX TESTING MOBILE APPLICATIONS USING SEQUENCE COVERING ARRAYS

An Undergraduate Research Scholars Thesis

by

ZACHARY B. RATLIFF

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisors:

Dr. Daniel Ragsdale
Richard Kuhn

May 2018

Major: Computer Science

TABLE OF CONTENTS

ABSTRACT

Black-box Testing Mobile Applications Using Sequence Covering Arrays

Zachary B. Ratliff
Department of Computer Science & Engineering
Texas A&M University

Research Advisor: Dr. Daniel Ragsdale
Department of Computer Science & Engineering
Texas A&M University

Research Advisor: Richard Kuhn
Computer Security Division
National Institute of Standards & Technology

Covering arrays have proven to be highly effective in detecting software bugs in what is known as combinatorial testing. A *t*-way covering array includes all *t*-way combinations of variable values, up to a specified level of *t* (usually 2-6 for software testing). In software systems that operate via a series of interactive inputs e.g. button clicks, a sequence covering array composed of sequences of events can be used. A *t*-way sequence covering array includes all *t*-way permutations of events (events are not required to be adjacent). This research examines the effectiveness of using sequence covering arrays to discover software bugs in mobile phone applications. Analysis of the distribution of *t*-way interactions between events in event sequence bugs provides insight into the practicality and usefulness of this combinatorial testing method. From a developer's perspective, these methods can contribute to finding this particular class of bugs early in the software development process, saving the developers time and money without sacrificing effectiveness. However, an attacker may also leverage these techniques to discover previously undetected bugs as a means to exploit the system. This method can be particularly

useful for attackers in that it is often simple to determine events in interactive software, even in black-box environments where internal knowledge about the source code is absent. Mobile applications running on popular operating systems such as Android and iOS are generally very interactive and therefore susceptible to these types of bugs. This project involved analyzing hundreds of software vulnerabilities in Android software, developing a new research tool for measuring sequence coverage in existing test suites, and using these combinatorial methods on various Android mobile applications.

DEDICATION

To my wonderful wife Kristen, my parents, Margaret and Weldon, my two brothers, Jake and Luke, and all my other friends and family that supported me throughout this endeavor. Thank you all.

ACKNOWLEDGEMENTS

I would like to thank my earliest mentors at the National Institute of Standards & Technology, Richard Kuhn and Raghu Kacker. They took me under their wing early in my college career and introduced me into the exciting world of combinatorics research in computer science.

Thank you to Dr. Daniel Ragsdale for taking a chance with me after just a few meetings of talking about my goals for this research project, and for always offering the best leadership and technical advice necessary to succeed. Thank you also to Dr. Trez Jones, Jennifer Cutler, and others in the Cybersecurity Center at Texas A&M who offered their help and guidance on the project.

I would also like to thank some of my earliest professors who helped inspire me to pursue a career in the field of computer science and engineering – Professor Rushikesh Hathi and Dr. April Andreas of McLennan Community College.

Finally, I would like to thank my wife, my parents, and all the family and friends who have continued to support me in everything that I do.

NOMENCLATURE

N	total number of tests
n	total number of parameters in tests
t	strength of covering array
v	number of values for each parameter
v_i	number of values for parameter *i*
CA(N, n, v, t)	*t*-way covering array of *n* parameters each with *v* values
S	finite set of events represented as symbols
SCA(N, S, t)	an N x |S| matrix where each t-way sequence of $s \in S$ appears at least once

CHAPTER I

INTRODUCTION

With the increasing threat of cyberattacks that governments, organizations, and individuals are experiencing each year, the call for more secure systems is louder than ever. A single flaw in the design, specification, or implementation of software can have costly consequences, and as a result there is a growing need for more proactive approaches in securing software. Advances in testing methodologies, software engineering practices, and vulnerability scanning has resulted in the decrease of severe software vulnerabilities being introduced into software, however, the overall number of vulnerabilities discovered each year is trending upwards [1].

One of the challenges that comes with securing software involves testing the vast input space for a given application. When a system has a large number of inputs it becomes increasingly difficult to test for correctness in all possible cases. Even with binary inputs, a modest number of parameters can make exhaustive testing intractable. Take for instance an encryption module with a 128 bit block size and a 256 bit key. To verify and validate the entire circuit would require testing 2^{384} total combinations of inputs. Granted the ability to test at a rate of one million tests per second, the sun would burn out long before the input space is exhausted. With software systems, the domain is often much larger since there are usually many ways to configure a system, along with testing all the input combinations for each configuration. There also arises the issue of parameters having a much higher number of possible values rather than being constrained to binary inputs. This challenge alone is partly responsible for a large portion

of software vulnerabilities that exist in the wild today, as companies simply do not have the time or budget to test the entire input space of their systems.

Combinatorial testing (CT) has proven to be a highly effective method for testing complex software systems with large input spaces [2][3]. CT tests all *t*-way interactions between input parameters rather than exhaustively testing the entire system. This drastically reduces the number of tests for any given system, making it much more practical in real world scenarios. One area of combinatorial testing that has garnered attention recently is in the concept of sequence covering arrays (SCA). Kuhn, et al. first proposed the sequence covering array in 2012, with the idea of finding bugs that are activated only after a specific sequence of events [4]. For example, suppose there exists a cloud based server application that accepts connections from different clients, and loads their saved media content after authentication. A foreseeable problem could arise from a specific sequence such as when a user logs into the server, loads a file, and then logs out before the file is fully loaded. Several errors can be proposed in this scenario, e.g. if the server only closes a connection after receiving an acknowledgement from the client that the media content was received. In this case a denial of service attack could be initiated using this sequence over and over again with many clients. Software bugs such as these are unique since their propagation depends on an order of events rather than a specific combination of inputs into the system. Testing for sequences such as these appear to be of higher importance in more interactive systems such as web applications, mobile applications, etc., however sequences exist in many other systems as well.

When testing software, it is sensible to use multiple methodologies in order to provide high quality code. Unit testing works well for test driven development, and stress testing can determine if the system can handle a copious workload, but both are necessary in their own right

to providing a stable system. Similarly, when searching for software vulnerabilities, several different attack vectors are often chained together in an attempt to get the system to behave in an unintended manner. Random large inputs are shoved into buffers, symbolic links are being altered, etc. sometimes simultaneously in hopes that the system will crash or output some information it's not supposed to. In similar fashion, sequence covering arrays are effective at triggering specific types of faults in software, however, when mixing SCA with other testing methods, one can uncover some very interesting security vulnerabilities.

In vulnerability research, it is regularly the case that the system being probed is a black-box. That is, the security researcher will have an idea of all the inputs and configurations feeding into the system, but will not have access to the actual blueprints i.e. source code, circuit schematics, etc. When faced with this challenge, various techniques are used to either try and better understand what is happening at the code level or to just try and cause a failure, e.g., reverse engineering and fuzzing. With many software systems, events can sometimes be deduced by simply sending inputs to the program or reading documentation on the architecture the software is running on (if available). For instance, consider commercial software running on Windows that connects to a URL provided as an input. Even if the source code is unavailable, one might be able to determine that the program is using the Winsock API or something similar for its TCP connections, and could decide that opening a socket, failing to open the socket, and closing the socket are all relevant events. Hackers may chain these events together in unexpected ways to produce the program failure they are looking for. If certain events are easy to determine without having much knowledge about the actual design of the system, then software developers would be wise to test for as many permutations of these events as possible. In the best case

scenario, developers will also test as many permutations of all events in the system, regardless of whether or not they can be determined in black-box scenarios.

Research Problem

The goal of this research project is to gauge the effectiveness of sequence covering arrays in discovering event sequence bugs in mobile applications. Particularly, in black-box environments where very little may be known about the underlying source code. In order to meet these goals, the following subproblems are explored:

- identify the distribution of interaction strengths for software bugs propagated by event sequences
- develop a tool that can measure the combinatorial coverage for existing event sequence test suites
- perform black-box testing on mobile applications using sequence covering arrays

Currently, there does not exist any data on the distribution of interaction strengths for event sequence bugs, nor are there many tools that support using sequence covering arrays in real world testing. By determining the upper-bound on the interaction strength of these bugs, software testers and vulnerability researchers can confidently decide the appropriate size of SCA to use for their scenario. There have been many significant studies on the *t*-way fault distributions in combinatorial interaction testing. Currently, combinatorial testing tools and covering array generators support values up to $t = 6$, as these studies have shown this to be the upper bound. A distribution of event sequence bugs would allow new and existing tools to be modified to support the upper bound for sequence covering arrays. The existence of an SCA measurement tool would allow for testers to decide whether or not to use an existing test suite based on if it covers an appropriate amount of *t*-way combinations. Finally, a study of these

methods in practice will provide further insight into the practicality of using SCA in black-box scenarios.

Solution

First, I analyze a collection of hundreds of software bug reports available from the NVD (National Vulnerability Database) and the Google Security Bulletin. Analysis includes studying the crash reports, code differentiation between the vulnerable code and non-vulnerable code fixes, and the interaction between events that propagated the bug. Gathering this information will lead to a distribution of *t*-way interaction strengths for these types of bugs. Furthermore, this distribution will contain an upper-bound for t, i.e. the highest strength sequence covering array for pseudo-exhaustive testing of sequences.

Second, these combinatorial testing methods are used in black-box environments. We select two different Android mobile applications and model sequence covering arrays after their events. Closed source 3rd party lock screens are chosen due to their inherent interactivity, as interactive software is believed to be more susceptible to event sequence bugs. Whether or not these methods are successful in finding newly undiscovered vulnerabilities will further determine if they should be used in real world scenarios.

Lastly, I develop a command line tool written in Haskell for measuring the coverage of sequence test suites. This tool will allow testers to measure the effectiveness of their existing test suites for finding event sequence bugs and help them determine whether or not an SCA should be used instead. The tool was written in Haskell to facilitate rapid development with a minimal code base. The tool will be continued to be developed and could eventually be ported to existing combinatorial measurement tools that don't yet measure for *t*-way interactions between events.

Contributions

This work makes the following contributions:

- The first ever distribution of *t*-way interactions between event sequence bugs is obtained. This distribution can be built upon as more and more software bug reports are analyzed. More bug reports should be analyzed in order to the further validate the upper-bound for *t*. If the value of *t* grows larger, e.g. $t > 9$, then the importance of efficient algorithms that can create large covering arrays also grows.
- A study on the practicality of black-box event sequence testing on mobile applications is performed. To our knowledge, this is the first study on event sequence bug testing in black-box environments that can be found in literature.
- The first ever tool for measuring the combinatorial coverage of event sequence test suites is developed. This tool will help facilitate using sequence covering arrays in real world software testing, as well as enable analysis of the benefit of switching to an SCA based test suite.

CHAPTER II

BACKGROUND

In this chapter we will discuss pertinent background information and related work in two main categories – combinatorial testing and Android mobile application security. Combinatorial testing requires the prerequisite knowledge of covering arrays, which will also be covered at the beginning of this chapter. All the background information covered here is related to the material that will be discussed in the remaining chapters of this thesis.

Covering Arrays

In mathematics, a covering array (CA) is an object that consists of all *t*-way combinations of variable values, up to a specified level of t. More formally, take the parameters $n_1, n_2, \dots, n_k$ with domains $D_1, D_2, \dots, D_k$. Then for any t domains, $D_i \times D_j \times \dots \times D_t$, there exists a row $r = (a_1, a_2, \dots, a_k)$ in the CA, with some *t*-tuple $(a_i, a_j, \dots, a_t) = T$, for all $T \in D_i \times D_j \times \dots \times D_t$. Contrary to an orthogonal array, which requires that each *t*-way combination of inputs appear the same number of times, a covering array's only stipulation is that each *t*-way combination appears at least once. Due to this leniency, covering arrays are much easier to construct than orthogonal arrays. In addition, a t-way covering array can always be found for any set of input variables and values. For some variable/value configurations, it is mathematically impossible to construct a t-way orthogonal array.

The "covering array number" is the number of rows that are associated with a given covering array. This number can be notated as CA(*t*,*k*,*v*) where t is the strength of the array, k is the number of parameters, and v is the total number of possible values for each parameter [5]. For example, CA(2,4,2) is a 2-way covering array spanning 4 binary input parameters. The size

of a covering array is not fixed i.e. rows can be added, however, optimality is usually preferred. An optimal covering array, denoted CAN(*t*,*k*,*v*), minimizes the number of rows in the array while still meeting the criteria of covering all *t*-way combinations of parameters. A general solution for the size of optimal covering arrays is still unknown. Figure 1 shows an optimal 2-way covering array with parameters.

0	0	0	0
0	1	1	1
1	1	0	1
1	0	1	1
1	1	1	0

Figure 1. Optimal 2-way Covering Array

For the covering array above, take a selection of any two columns, and every combination of 0 and 1 will be present (00, 01, 10, 11). The compactness of the covering array is what makes it useful. In only five rows, all twenty-four 2-way combinations of parameter values are covered. If we were to exhaustively cover all combinations of the parameters, sixteen rows would be needed. The relationship between covering array size and (t, k, v) depends on the algorithm used to construct the covering array. Existing algorithms used in practice construct covering arrays that grow exponentially to t, but logarithmically to k.

$$\text{Covering array size} \propto v^t \log(n)$$

It should be noted that it is possible to create mixed strength covering arrays as well, i.e. a covering array in which each parameter contains a different number of values. This is useful in practice since most systems contain more complex inputs such as alpha-numeric sequences. The total number of *t*-way combinations of inputs for such systems follow the equation:

$$\sum_{i \neq j \neq \cdots \neq t}^{n} v_i \times v_j \times \ldots \times v_t\,, \qquad \forall \binom{n}{t} \text{ combinations of parameters}$$

where v_i is the number of possible values for parameter *i*, and *n* is the number of parameters in the system. The summation is over all $\binom{n}{t}$ combinations of parameters in the system.

Combinatorial Testing

The power of covering arrays become apparent when constructing test suites for large software systems. With any piece of large software, testing all combinations of inputs quickly becomes intractable. In certain life-critical situations, having high confidence that the software will not fail is imperative. Consider an autonomous vehicle transporting human passengers. As it is traveling down the highway at 60 miles per hour, a software failure in the collision avoidance system could cause serious injury or even the loss of life. But how can the engineers ever have high confidence that the system will not fail if they are only able to test a small fraction of the possible inputs? Combinatorial testing aims to solve this problem by choosing specific test cases which are statistically more likely to trigger the bugs, if any, that are present in the software.

Previous research has analyzed thousands of software bug reports from various software systems to determine a distribution of the interaction strength in software bugs. Specifically, how many parameters contribute to the fault propagation of a given bug. Empirical evidence gleaned from the analysis suggests the *interaction rule*. That is, most software failures are caused by one or two factors, with progressively fewer caused by 3 or more factors [2]. Figure 2 illustrates the interaction rule and demonstrates how covering arrays could be useful in software testing.

Interaction Rule: Most failures are induced by single factor faults or by the joint combinatorial effect (interaction) of two factors, with progressively fewer failures induced by interactions between three or more factors.

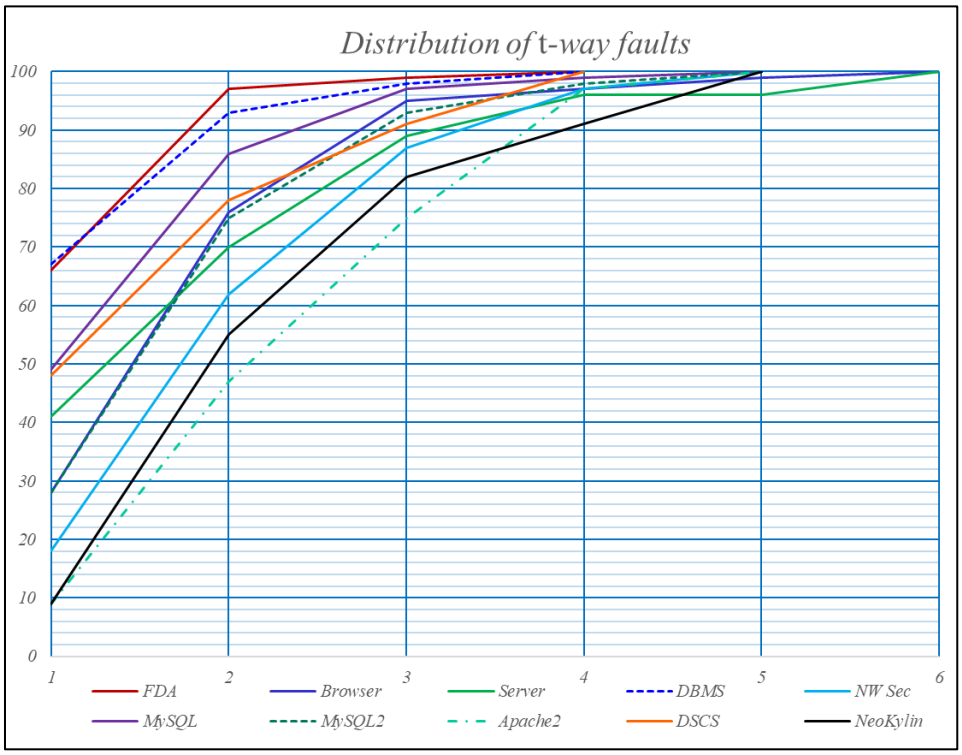

Figure 2. Distribution of *t*-way faults

Out of all the software bugs analyzed, no failure was induced by a more than 6 factor fault [2][6]. Furthermore, high confidence in finding most software bugs can be achieved by testing for all 4-way combinations of input parameters. Based on this finding, combinatorial testing proposes to use *t*-way covering arrays modeled after the system as test suites. Since 6-way covering arrays would be sufficient for finding all of the software bugs from those analyzed in past studies, software testers can have high confidence in finding all the software bugs in a system when using covering arrays of similar strength, with an appropriate input model that uses representative values for variables with large domains (discussed below).

Consider a simple mobile phone application which can run on both iOS and Android. Let's suppose the application is used for taking and sharing photos with social media friends. The application may have several different inputs during runtime e.g. description of a photo, phone number, username, etc. However, assume a scenario where the developers want to ensure that the application functions correctly in all configurations of the system. In the real world, the total configurations is likely very large and complex, but a simple example will be used here. Table 1 shows the various configurations for the application in this scenario.

Table 1. Application Configuration Example

Platform	**hasCamera**	**acceptedContactsPermission**	**lowBattery**	**numberOfFriends**
Android 5.1	true	true	true	0 to $2^{31} - 2$
Android 7.0	false	false	false	$2^{31} + 1$ to -1
…				$2^{31} - 1$
iOS				2^{31}

Notice that the *numberOfFriends* configuration parameter only contains four possible range values, rather than all the integer values that it could actually take. Reducing the input space only to values that are logically different i.e. cause the path of execution to change, is common in software testing. The resulting ranges are known as equivalence classes, and any value within a given equivalence class is assumed to have the same effect on execution as other values within the same class.

Now, assume there are a total of 10 different platforms that the application is intended to run on. This yields $10 \times 2^3 \times 4 = 320$ different system configurations. The only way to actually

ensure that the system functions correctly in all cases is to test all 320 configurations. However, the developers may come to the conclusion that testing every 2-way combination of configuration parameter will be sufficient for their purposes. Current combinatorial testing tools can generate 2-way covering arrays of this system with fewer than 100 rows, reducing the number of tests by more than 220! In practice, the reduction in the number of tests when using *t*-way covering arrays is often much more drastic, resulting in a significant amount of savings on time and money during testing.

Sequence Covering Arrays & Event Sequence Testing

Combinatorial testing methods have proven to be effective in finding software bugs in various systems. However, the methods described in the previous section are not useful when trying to detect software bugs that propagate via a series of events. Using our mobile application example from the last section, let's suppose the application fails when a user attempts to share a photo after denying the application the necessary permissions it needs to access the user's contacts. Simply setting the *acceptedContactsPermission* parameter to false is not enough to trigger the failure, since a particular order of events is required.

When a particular order of events in software can cause failures, a different type of covering array is needed for combinatorial testing. Kuhn, et. al proposed the sequence covering array in 2012 as a means for detecting these *event sequence bugs* [4]. Sequence covering arrays differ from ordinary covering arrays in that they are defined as an N x S matrix where entries are from a finite set S of s symbols, such that every t-length permutation of symbols from S occurs in at least one row. Furthermore, the t symbols in the permutation are not required to be adjacent. In software, events typically lead to some change in state. A later failure may depend on whether or

not that state in the program has previously been reached, but not necessarily immediately before the failure-triggering event.

For example, consider the events *a*,*b*,*c*,*d*, and *e*. There are $5! = 120$ permutations of these events. However, Figure 3 shows that all 2-way permutations of events can be covered in only two tests.

a	b	c	d	e
e	d	c	b	a

Figure 3. 2-way sequence covering array

Pick any two events from the set $S \coloneqq \{a, b, c, d, e\}$, and both permutations of those events are present within the SCA in Figure 3. Indeed, for any set of events, the optimal 2-way sequence covering array only contains two tests: the set of events, and the reverse order of this set. However, more intelligent algorithms are needed for generating *t*-way sequence covering arrays with $t > 2$. Figure 4 below shows a 3-way sequence covering array for the same set of events.

a	b	c	d	e
e	d	c	b	a
c	d	e	a	b
b	a	e	d	c
d	a	c	b	e
e	b	c	a	d
c	a	e	b	d
d	b	e	a	c
a	e	d	c	b
b	c	d	e	a

Figure 4. 3-way sequence covering array

Now pick any two events from the set $S := \{a, b, c, d, e\}$, and all $3! = 6$ permutations of events will be present within the SCA. For instance, *abc*, *acb*, *bac*, *bca*, *cab*, and *cba* are in tests 1,5,4,6,3, and 2 respectively.

Android Activity Lifecycle

Understanding various aspects of the Android operating system as well as how individual Android applications are constructed will be useful during test suite generation. Having considered events at the operating system level and how they can affect a top level application will allow for a much more robust system model. In Android, the basic building block for an interactive application is the "Activity," which provides several callback methods that monitor and react to its own state. This section will examine the Android Activity lifecycle and discuss how the transitions through this lifecycle are useful when modeling application events.

The Android Activity lifecycle consists of seven different callback methods that are called as the state of the activity changes [7].

- **onCreate** – called when the system first creates the activity; includes basic application start up logic that is only called once throughout the activity life cycle
- **onStart** – called when the activity first starts; prepares the activity to be presented to the user and often contains initializations for code that maintains the user interface
- **onResume** – called when the activity enters the foreground; the user is interacting with the activity when it is in this state
- **onPause** – called when there is some indication that the user is leaving the activity i.e. if the activity moves out of focus; code that manages and releases system resources is often included in this callback

- **onStop** – called when the activity is no longer visible to the user e.g. when a user multitasks and moves a separate application's activity to the top of the activity stack; this callback will generally include more CPU intensive operations such as saving data to a database before the application is destroyed
- **onRestart** – called first when the activity is recovered after entering the *onStop* callback
- **onDestroy** – final call before the application is destroyed; only called once

These callback methods generally contain some sort of side effect such as data initialization, garbage collection, etc. For this reason, the Android Activity lifecycle can be included in our system model of events with each callback method representing a different event in the system. In chapter 3, we will see that it may not be necessary to explicitly include these callback methods as events if they are already implicitly included in our test suite.

Reverse Engineering Android Applications

The term "black-box" implies not knowing anything about the internal workings of a system. The inputs and outputs of the system are known, however, how the inputs are manipulated to produce the given output is a mystery. Hackers and security professionals alike are often faced with this problem, and several techniques are used as a means to gain insight into the core processes of the system. If the software is easy to reverse engineer, then not having the source code becomes trivial.

Software programmed in Java and compiled with the Java compiler are notoriously easy to reverse engineer. Many disassemblers exist that can produce a near perfect recreation of the original source code from Java bytecode. From a hacker's perspective, this means having access to the .jar file alone is usually enough to go from a black-box scenario to a more favorable white-

box scenario. Although obfuscation techniques can be employed in the source, given enough time and motivation, a reverse engineer will be able to deduce the underlying logic.

Most Android applications are programmed using the Java programming language. However, the compilation process is slightly different on the Android platform. Java code is first compiled into Java bytecode using the Javac compiler. This bytecode is then converted to DEX (Dalvik Executable) code, using the Android DX compiler. DEX code is similar to Java bytecode in that both are executed by a virtual machine, however, DEX code runs on the Dalvik VM, whereas Java bytecode is executed on the Java VM. There were several reasons for Android deciding to use their own version of a Java virtual machine that won't be discussed here, but it essentially boils down to memory and processing constraints on mobile devices. From a reverse engineering perspective, disassembling DEX code into a near perfect representation of the original source code is not much different.

Recently, the Android platform switched from the Dalvik virtual machine to ART (Android RunTime). ART brought significant improvements, specifically in how applications are executed. Instead of using Just-in-time (JIT) compilation (the Dalvik VM coverts DEX code to machine code at runtime), ART uses Ahead-of-time (AOT) compilation (the DEX code is converted to native machine code just once upon install). This speeds up execution of applications, primarily during startup. The switch doesn't affect the disassembly process however, since the original DEX code is included in the new ART .oat files.

So if it is fairly simple to reverse engineer Android applications, why do we focus on black-box scenarios in this research? First, the insight gained from this research will apply to other areas in mobile security such as testing iOS applications. Second, we believe that new techniques in obfuscation and other advances on the Android platform will continue to make

reverse engineering more difficult and time consuming, therefore prolonging the vulnerability discovery process. Lastly, there are several scenarios where reverse engineering the Android application provides little to no clues about the underlying logic. For instance, to defend against reverse engineering, some developers are moving what used to be client-side computations over to a server. In this case, the system as a whole (the Android application and the server application) are a black-box that cannot be completely reversed.

CHAPTER III

METHODOLOGY

In order to gauge the effectiveness of using a combinatorial test suite design to trigger event sequence bugs in software, the distribution of interaction strengths needed to be determined. For instance, if it was discovered that the upper bound on the interaction strength of an event sequence bug was eight, then an 8-way sequence covering array would be needed to provide high assurance during testing. This distribution would be provided by analyzing a large amount of software bug reports from different vulnerability databases. Furthermore, two separate 3rd party Android applications are tested using sequence covering arrays. The applications tested were first modeled based on the observed events within the application, and then several constraints were removed in order to accommodate lazy SCA generation e.g. ignoring adjacent events. The effectiveness of this approach is discussed more in Chapter 4. Lastly, we develop a combinatorial coverage measurement tool for event sequences, CSCM (Combinatorial Sequence Coverage Measurement). The performance of this tool is evaluated by measuring several practical sized test suites.

Android Vulnerability Analysis

The Android Security Bulletin is a monthly bulletin dedicated to providing the latest information on security patches for the Android Open Source Project (AOSP), the Linux kernel used in Android, and other driver software present on Android devices [8]. Android device and chipset manufacturers also keep bulletins of various security patches that they apply to their proprietary software, and the Android Security Bulletin will generally contain links for more information on these patches as well. Each month a new bulletin is released, describing the fixes

that have been implemented for newly discovered vulnerabilities. Note that this vulnerability database does not include vulnerabilities from 3rd party applications like ones that could be downloaded from the Google Play Store, but instead only contains information and vulnerabilities pertaining to pre-installed applications i.e. system applications that are part of the Android operating system, or pre-installed device manufacturer applications. Nonetheless, the Android Security Bulletin contains thousands of patched software vulnerabilities, often with vivid descriptions of the vulnerability itself, and the source code changes that were implemented to apply the fix. Using the description of the vulnerability along with the changes that are reported in the source code, we can generally make a confident assessment in the type of bug we are analyzing e.g. event sequence bug, memory corruption, etc. along with the various factors that contribute to its propagation.

Another useful vulnerability database is the United States National Vulnerability Database (NVD) operated by the Information Technology Laboratory at the National Institute of Standards & Technology (NIST). The NVD contains an abundance of information regarding software flaws, misconfigurations, product descriptions, and impact metrics for numerous products, including the Android Operating System [9]. The database is populated by CVE (Common Vulnerabilities and Exposures) which is a list of vulnerabilities, each containing an identification number, public references, and description of the flaw. CVE defines a vulnerability as "*a weakness in the computational logic (e.g., code) found in software and hardware components that, when exploited, results in a negative impact to confidentiality, integrity, or availability. Mitigation of the vulnerabilities in this context typically involves coding changes, but could also include specification changes or even specification deprecations (e.g., removal of affected protocols or functionality in their entirety)*" [10]. By this definition, what a developer

might characterize as a software bug, may not be characterized as a software vulnerability and therefore wouldn't be listed in the NVD. For instance, consider a software flaw that incorrectly formatted text at the top of a mobile application. Although this may be considered a bug that needs a fix by the developers, it doesn't actually constitute a negative impact to confidentiality, integrity, or availability. However, take the same scenario, except now the bug inadvertently displays the user's credentials on the screen. This would be considered a vulnerability, and depending on the product, may get its own CVE.

All of the software vulnerabilities listed in the Android Security Bulletin are also listed in the National Vulnerability Database. However, both databases were used as the NVD often contained links to other sources that were useful. Frequently, it was the case that both databases contained insufficient information to properly determine the vulnerability's propagation. Even with the source code changes in hand, without expert knowledge of the underlying component or a more detailed description of the vulnerability, it was often too difficult to make an accurate assessment of how one might trigger the failure. Some vulnerabilities contained multiple sources outside of the Android Security Bulletin and NVD, describing their propagation and exploitation. These sources were also considered during our analysis, however, much more weight was given to descriptions in the bug databases as they are verified by the Android security team. The internet presence of proofs of concept for different vulnerabilities were in large part the main factor in determining the validity of these outside sources. That is, if an exploitable bug contained multiple proofs of concept and supporting descriptions from different sources outside the Android Security Bulletin and NVD, then we were more likely to consider the information from the outside sources. However, if a specific vulnerability had a smaller internet presence i.e.

one un-verifiable proof of concept, then only the descriptions from the Android Security Bulletin and NVD would be considered.

The distinction between vulnerability and bug has been made, but we should also emphasize that the vulnerabilities that we are considering are those that can be discovered via event sequences, but not strictly caused by event sequence bugs. Vulnerabilities may exist without the presence of a bug e.g. a system can be implemented perfectly according to specification, but this does not mean that it is free from negative impact on the confidentiality, integrity, or availability of the system. In the context of this research, a vulnerability may be discovered by a particular event sequence, without the presence of an event sequence bug. Consider an FTP application that stores the plaintext passwords in a text file on the system. This may be exactly what the developers had in mind, and therefore it wouldn't be considered a bug, however, it is considered a vulnerability. From a black-box testing perspective, if we can discover this vulnerability via a particular sequence of events, then we will account for the interaction strength in our distribution. There are two reasons behind including these vulnerabilities in our distribution of event sequence bugs. First, is that most vulnerabilities of this type are eventually considered design or specification bugs, i.e., there was a flaw in the way the system was originally specified to behave. Second, from a black-box testing perspective, this is usually the exact type of thing a tester or hacker is looking for.

The software vulnerabilities we decided to analyze were all the vulnerabilities listed in the Android Security Bulletin from August 2015 to December 2017, excluding those pertaining to device or chipset manufacturers. An example of such a vulnerability is CVE-2016-2457. This particular CVE entry existed in the WiFi module and was described as an elevation of privilege vulnerability: "*An elevation of privilege vulnerability in Wi-Fi could enable a guest account to*

modify the Wi-Fi settings that persist for the primary user. This issue is rated as Moderate severity because it enables local access to 'dangerous' capabilities without permission". From the description alone, one might be able to deduce that running a specific sequence of three events would find this vulnerability:

1. Switch to guest account
2. Modify Wi-Fi settings
3. Switch back to the primary user

Execution of these events in order would allow a tester to discover the dangerous vulnerability. Note that these events are not required to be executed immediately after one another. If this were the case, then the construction of our test cases would be much more restricted and sequence covering arrays would not suffice (as sequence covering arrays do not require the events be adjacent). When analyzing the software bugs, we were sure to make this distinction so that the actual effectiveness of using sequence covering arrays could be determined. CVE-2016-2457 was decided to have an interaction strength of 3 – that is, a 3-way sequence covering array would trigger the sequence of events needed to discover the vulnerability. More often than not, the description is not enough to make a confident analysis of the vulnerability. In the case of this vulnerability much more information was also provided in the change log, along with the source code differentials. Figure 5 shows a short example of what one might expect to see in the change log for a given bug on the Android Security Bulletin.

[DO NOT MERGE] Disallow guest user from changing Wifi settings

Disallow existing and newly created guest users from
changing Wifi settings.

BUG: 27411179
TEST: Flashed device, switched to existing guest user, and verified
that Wifi settings are disabled.
TEST: Flashed device, created new guest user, and verified that Wifi
settings are disabled.

Figure 5. CVE-2016-2457 Change log

Although it may not seem like much information, we can deduce what events would discover the vulnerability if it still existed by examining the test cases used in verifying the fix. In the first test case it states "*Flashed device, switched to existing guest user, and verified that Wifi settings are disabled.*" From this description, we can see that the developers likely did not intend for WiFi settings to be available for guest users, as this option was completely disabled in the fix. Furthermore, a quick survey of the source code changes in Figure 6 show the new restriction on guest accounts:

```
    if (mGuestRestrictions.isEmpty()) {
        mGuestRestrictions.putBoolean(UserManager.DISALLOW_OUTGOING_CALLS, true);
        mGuestRestrictions.putBoolean(UserManager.DISALLOW_SMS, true);
+       mGuestRestrictions.putBoolean(UserManager.DISALLOW_CONFIG_WIFI, true);
    }
```

Figure 6. CVE-2016-2457 Code Differential Snippet

Whether or not this was a specification bug or an implementation bug is not known, which brings up an important point in our discussion on event interaction classification. If it is an implementation bug (i.e. the developers originally allowed WiFi settings on guest accounts, but

didn't want the changes to persist upon switching accounts), then the 3-way sequence covering array would be needed for vulnerability discovery. However, if it is a specification bug (i.e. the specification did not correctly state that WiFi should be disabled completely on guest accounts), then only two events would be needed to find the flaw: 1.) switch to guest account, 2.) access WiFi settings. In these types of scenarios, re-examining the actual vulnerability details is helpful. Is the vulnerability the issue of guest users having access to WiFi settings? Rather, is it that the guest user has the ability to modify the WiFi settings of other accounts on the system? For this vulnerability, we consider the latter to be the correct choice as the original description of the vulnerability describes it as a privilege escalation. In other analysis where too much ambiguity is involved to make a confident decision, we take the interaction strength to be the upper bound of the choices. By choosing the upper bound we can ensure that testing based on this data will use sufficiently long *t*-way sequences.

Several vulnerabilities analyzed had more complex characteristics pertaining to the event sequences that caused them. CVE-2015-6643, was described as being a factory reset protection bypass in the setup wizard. Factory Reset Protection (FRP) was employed in Android to prevent access to stolen devices via prompting the user to enter the password associated with the Google account on the device, before performing a factory reset. In many cases where mobile devices are stolen, the phone is rendered useless due to a screen lock that prevents the criminal from accessing the device. If access is not prevented, the FRP mechanism deters the criminals by preventing factory resets which wipe the phone's memory completely. A primary motivator for stealing mobile phones is selling the device for money. Modern day cell phones can range upward in prices of $900 to $1,000+, which is a huge incentive for criminals. However, many people will refrain from buying devices that appear to be stolen, even at large discounts. Since

the owner's data cannot be removed from the phone without knowing the password that was last used to access the device, making the phone appear brand new becomes a challenge. The FRP bypass exploit was achieved via a sequence of events in the setup wizard:

1. hit button to access Google keyboard settings (this is usually achieved by holding down the '@' or ',' keys)
2. Select "Help & Feedback"
3. Long press on any text on the screen (this brings up another dialog)
4. Click "Search" and search for "settings"
5. Now in the settings menu, click "About Phone"
6. Tap the "Build number" icon several times to enable developer options
7. Go to developer options and disable FRP (OEM unlocking) to factory reset device without FRP

In this case, the vulnerability was fixed by disallowing text search in the setup wizard (step 4). Although the exploitation required seven steps, the actual vulnerability (allowing access to the settings) occurred after only four steps. Furthermore, a 4-way sequence covering array may not be useful here since most of the events are adjacent i.e. can only be exercised immediately following another event. For instance, selecting "Help & Feedback" can only be exercised after accessing the google keyboard settings. We will discuss in the next section a lazy method for dealing with adjacent events in which a 4-way SCA would suffice for discovering this vulnerability.

Admittedly, vulnerabilities such as CVE-2015-6643 are inherently more difficult to discover even with sequence covering arrays. The vulnerability appears to be introduced by the developers not knowing that the settings could be accessed from the keyboard. This could be due

to several factors such as one team working on the keyboard functionality, and another team working on the setup wizard functionality. The setup wizard team may not be aware of the full specifications of the keyboard software and therefore wouldn't know of all the possible events. Knowing of the existence of all possible events in the software is critical in the construction of sequence covering arrays.

One last example of the analysis conducted in this research will demonstrate a vulnerability that is not classified as being caused by event sequences. Often times we found that event sequences may be involved, however, the root cause of the vulnerability was due to improper handling of data. If data is also involved in the bug / vulnerability propagation, then a sequence covering array alone would not be sufficient in discovering it. CVE-2017-0593 is described as a privilege escalation vulnerability in the Framework APIs. A comment in the code describes the fix as well as the initial cause of the vulnerability: "*Prevent apps to change permission protection level to dangerous from any other type as this would allow a privilege escalation where an app adds a normal permission in other app's group and then redefines it as dangerous leading to the group auto-grant.*" Several events are listed here and further analysis of the source code is needed for proper classification. However, from this comment alone we can see that part of the issue is caused by the actions "change permission" and "redefine permission." Although a tester could generate an SCA based on such events, specifying these alone would not be adequate, as it is the actual data (permission type) that is the origin of the vulnerability. For example, the comment states that the permission change must be from any type to the specific "dangerous" permission type. So changing permissions to any type other than "dangerous" would not contribute to the vulnerability's propagation. Events could be defined such as *changePermissionToDangerous*, however, if a large number of permissions existed, then this

would be extremely impractical to do for all of them. For this reason, vulnerabilities such as this were not considered in our analysis.

Black-box Testing 3rd Party Android Applications

Analyzing the distribution of interaction strengths between event sequence bugs provides valuable insight into whether or not sequence covering arrays are a viable option for bug discovery. Testing the effectiveness of SCA in actual black-box scenarios would help further gauge the practicality of this technique. This process would involve selecting a couple of closed source 3rd party Android applications and generating test suites modeled after the system's events. In chapter 2 we discussed the simplicity of reverse engineering many Android applications, however, no such attempts were made on the applications chosen, both for legal reasons as well as preserving the black-box environments needed for this research.

After analyzing several of the vulnerability reports in the Android Security Bulletin, access control seemed to be an interesting area of application functionality to experiment with. Several of the software vulnerabilities such as CVE-2015-6643 bypassed access restrictions simply by exercising a series of events. Since many of these vulnerabilities were present in the Android system itself, we would expect several 3rd party application software to make similar mistakes. Many applications aren't actively developed and may contain security vulnerabilities uncovered from the newest techniques in vulnerability research. Furthermore, some applications are fairly new and will not have been scrupulously tested by a large userbase.

The specific type of application we focused on were 3rd party lock screens. Vulnerabilities in these applications have serious implications for the overall security of the system. If a lock screen application is bypassed by an attacker with physical access to the device, the entire system can be compromised. Since security is the main functionality that should be

provided by a lock screen, exploiting this feature can have a very negative impact on the overall perception of the software. When a user installs a 3rd party lock screen application, the additional functionality of the app is what is generally most appealing. For instance, many of these applications will include several different widgets, tools, or background designs that can be accessed from the lock screen, which are not available on the stock Android lock screen. However, when a user downloads this software, they will expect the same level of security that the stock version supplies.

Since the Android operating system runs on a wide variety of devices from various manufacturers, it is expected that certain event sequence bugs will be present on some devices but may not be present on others. The same holds for various versions of the Android OS. For instance, Android 7.0 includes the ability to call an emergency contact from the lock screen, however, Android 5.1.1 does not include this functionality. From a developer's perspective, these events may not be accounted for if the application hasn't been updated to run on the latest version of Android. For this research, we decided to use three different phone models running different versions of the Android operating system. The devices we used for testing the chosen 3rd Party lock screen applications are listed below.

- Sony Xperia Z1 (Android 5.1.1)
- Google Nexus 6P (Android 7.0)
- Sony Xperia XZ Premium (Android 8.0)

The first application we chose to perform testing on was the Microsoft Next Lock Screen. The Microsoft website says the following about the application: "*Next Lock Screen is the ultimate lock screen for busy professionals. Next helps protect your phone from unwanted access by others, and you are instantly more productive. It is an excellent productivity locker app and*

companion for your everyday life" [12]. The Google Play Store listing also lists several accolades for the software such as AndroidPIT's 2016 "Best Lock Screen for Android" award. Although Microsoft Next has over 1 million installs and appears to be a popular choice among users looking for an alternative lock screen, the last update occurred over one year ago, leading us to believe that it is not currently in active development. Nonetheless, the app has several notable features that would be appealing to users looking for an alternative lock screen solution.

- Custom unlocking via a PIN or pattern
- View missed calls, text messages, and notifications from popular social media websites
- Launching applications with a built in launcher straight from the lock screen
- Quick access to tools such as the camera, flashlight, WiFi, Bluetooth, and more

There are several events that can occur within the context of the Microsoft Next Lock Screen. Modeling our test suite to include every event possible will improve our ability to find new bugs and vulnerabilities. A quick glance at the user interface of the lock screen application gives clues as to what actions are available to be performed. As shown in Figure 7 below, there appears to be functionality for recovering a forgotten PIN / Pattern as well as opening the camera. This information is obtained simply by observation and shows the little amount of effort involved in identifying events within interactive software.

Figure 7. Microsoft Next Lock Screen Application

Some events within the Microsoft Next Lock Screen are not apparent by observation, but are instead system wide events offered by the Android operating system. For instance, many versions of Android come with the ability to activate the Google Assistant by speaking the phrase “OK Google.” The Google Assistant software can turn on the device’s flashlight, perform web searches, call contacts in the user’s phonebook, etc. Events such as these can have serious security implications if not handled properly by lock screen applications, so we will include them in our test suite. Figure 8 below lists all of the events of interest we have identified for this lock screen.

Event	Description
A	Push soft camera button
B	Push hardware camera button
C	Voice activate google assistant
D	Activate google assistant via home button long press
E	Enter Pin
F	Change wallpaper
G	Send forgot PIN/Pattern email
H	Click Widget shortcuts
I	Take screenshot
J	Notification bar settings button
K	Accept Phone Call

Figure 8. Events of interest in the Microsoft Next Lock Screen Application

The events in Figure 8 are fairly general. In practice, they may be as abstract as needed to test any given functionality. As indicated in chapter 2, Android activities contain several different callback functions that are called during different stages of the activity lifecycle. If we wanted to include the *onResume* function for the main activity of the Microsoft Next Lock Screen, we could list it as an event in our table as well. Often these Android activity functions are indirectly included in our sequence covering array as we transition between executing the various events. For instance, if "hit multitask button" is an event, then every time we execute this action, the activity's *onPause* function will also be called (since *onPause* is called every time an activity is no longer in focus in the foreground). When we return to the activity, the *onResume* function will be called as a result.

As mentioned earlier, some of these events are only available on certain versions of the Android operating system as well as certain devices. Open camera via hardware button would not be available on the Google Nexus 6P, so we would ignore this event when testing on that device. Separate sequence covering arrays can be generated based on which system is being

tested. Using the combinatorial sequence test generator tool developed by NIST, we constructed a 4-way sequence covering array based on the above events. This was the maximum size SCA offered by the NIST tool, and the overhead incurred by generating a 4-way SCA versus a 3-way SCA is negligible in this system model.

Note that previous studies have shown sound methods of modeling interactive systems, specifically for generating *t*-way test suites for event sequences [11]. Since there are many caveats when considering event sequences (adjacent events, constraints, etc.), these methods are needed for an accurate constructions of test suites. However, we are approaching this research from a black-box perspective, and will instead use a lazy approach that uses the least amount of knowledge about the system as possible for test suite generation. The lazy approach only considers independent events from the context of the current activity, and excludes transitions. For example, if the camera can be opened from screen A, and it can also be opened from screen B, these will be treated as two separate events. Overall, this greatly increases the number of events in our system model, however, since covering array size only grows logarithmically to the number of events, the resulting increase in sequence covering array size is small. Furthermore, consider the following:

- A system containing events *a,b,c,d*
- *b* can only be executed after *a*

Now suppose that a 3-way SCA is generated which includes the test *b,c,a,d*. Since event *b* cannot be executed until after event *a*, the test can be modified to be: *a,b,c,a,d*. Note, that event *a* is executed twice in this test, but it fixes the constraint violation we faced with minimal effort. In practice, our tests may contain many constraint violations and this method can greatly increase the average length of our tests.

The second Android lock screen application tested during this research is Super Locker. The popularity of this lock screen was the primary reason for choosing it in our tests. With over 5 million downloads, and nearly 50,000 reviews, the application is one of the most downloaded lock screens in the Google Play Store. Super Locker also appears to be in active development as the last update occurred just one week before downloading it for testing. Although there are several lock screen applications that we could have chosen, Super Locker offered a wide variety of features such as a music control center, weather widgets, and camera access [13]. A lock screen with many features would likely be more exciting as there is a larger set of event sequences that can occur.

Modeling the Super Locker application followed the same observational process that was used when modeling the Microsoft Next Lock Screen. There were a few more notable events that are available within the Super Locker application, however. For instance, the application contains a “boost” button which will free system resources that are slowing down the phone. Advertisements, widgets, weather, and music also appear in a scrollable window that is accessible from the lock screen. One of the more interesting differences between Microsoft Next and Super Locker, is that Super Locker supports making emergency calls from the lock screen. Figure 9 on the next page displays the events of interest in the Super Locker application.

Event	Description
A	Push soft camera button
B	Push hardware camera button
C	Voice activate google assistant
D	Activate google assistant via home button long press
E	Select Application Widget
F	Click Advertisement
G	Boost Device
H	Enter Pin
I	Emergency Call
J	Notification bar settings button
K	Take Screenshot
L	Receive phone call

Figure 9. Events of interest in the Super Locker Lock Screen

Earlier in the chapter, we mentioned that events would be independent and considered in the context of each activity. This would mean that event *B* could be labeled as multiple events (push hardware camera button from lock screen, push hardware camera button from news feed, etc). However, in this case it makes sense to label the "push hardware camera button" as a single independent event, since it can occur from every state in the program i.e. there are no constraints on the event. Now consider event *A*. Opening the camera via the soft camera button can only occur from the main activity in the lock screen, so this is the only event corresponding to the soft camera button. However, if the soft camera button appeared on multiple screens (but not all), each would be labeled as separate events.

CSCM Tool

Several tools exist that help facilitate combinatorial testing in practice. For this research, we generated sequence covering arrays using the NIST sequence covering array generator tool. Other tools such as ACTS (Automated Combinatorial Testing for Software) are used for generating ordinary covering arrays for interaction testing, while tools such as CCM (Combinatorial Coverage Measurement) measures the coverage of these generated test suites. Currently, there does not exist a tool for measuring the combinatorial coverage of sequences in test suites. This research aimed to develop such a tool which would be useful for event sequence testing as well as comparing sequence covering array generation algorithms.

The Haskell programming language was used to develop CSCM (Combinatorial Sequence Coverage Measurement). Haskell is a pure, functional programming language that required a minimal amount of source code and enabled rapid development. Many libraries exist with efficient algorithms for creating permutations, sorting lists, removing duplicate elements, etc. Counting *t*-way sequences can be computationally intensive, especially for large test suites with long tests (20+ events per test), so parallel processing was needed for eliminating many bottlenecks. For instance, each test contains k number of *t*-way sequences, and counting these subsequences can be performed in parallel since they are independent of each other. Performance metrics for the CSCM tool are presented at the end of chapter 4 and highlight the performance boost that is incurred when utilizing multiple processors.

The algorithm for computing *t*-way coverage is quite simple, and most of the challenges in developing the CSCM tool were in computational complexity. Below are the general steps involved with creating any sequence coverage measurement tool:

1. Count the total number of *t*-way sequences for a given set of events E.

$$e = \binom{|E|}{t} \times t!$$

2. For each row in the test suite, store the *t*-way subsequences in an array.
3. Remove all duplicates from the array yielding the total number of unique *t*-way sequences in the test suite, *k*.
4. Divide the unique number of *t*-way subsequences in the test suite by the total number of possible *t*-way subsequences.

$$coverage = \frac{k}{e}$$

This general algorithm was used when developing CSCM, however, step 2 is performed in parallel if the user has multiple cores available. The other bottleneck that exists is in step 3, which has complexity $O(n^2)$, however, we achieved $O(nlog(n))$ complexity utilizing an ordered list implementation available in the *Data.List* Haskell package.

CHAPTER IV

RESULTS & IMPLICATIONS

In this chapter will discuss the results from our three experiments. Mainly, the distribution that was obtained via analyzing event sequence bugs in Android, the black-box testing results on 3rd party lock screen applications, as well as the performance of our newly developed combinatorial sequence coverage measurement tool. The implications of such results will be discussed here in detail.

Vulnerability Analysis Results

The Android Security Bulletin contained hundreds of vulnerability reports from August 2015 to December 2017. Initially, all software vulnerabilities listed in the bulletin were considered. However, after investigating the content of several of the reports, the search space was reduced to vulnerabilities which contained links to the Android source code. These were generally AOSP specific vulnerabilities, but would sometimes be device specific e.g. on the Nexus mobile phones. Vulnerabilities related to external components such as the Linux kernel, or Qualcomm drivers either didn't contain enough information for confident classification, weren't related to events, or were closed source (as in the case with many of the Qualcomm component vulnerabilities). For instance, many of the Linux kernel vulnerabilities analyzed early on were related to memory or bounds checking issues such as use-after-free, integer overflows, etc. It certainly seems that kernel vulnerabilities are much less likely to contain event sequence bugs than vulnerabilities that occur at higher levels in the system. One reason for this could relate to interactivity, i.e., the more interactive a component, the more likely it is to contain event sequences that lead to vulnerabilities.

After analyzing over two years' worth of Android vulnerabilities, only a small fraction were determined to be related to event sequences. While 592 vulnerability reports were analyzed, only 49 (approximately 7.9%) were determined to be caused by some sequence of events. Although this number appears to be quite small, it is not insignificant. All of the vulnerabilities caused by event sequences were rated from "Moderate" to "Critical," and therefore could be an attractive target for attackers. Furthermore, if event sequences aren't being considered at all during testing, it is likely that a significant number of vulnerabilities could exist within the code base. As mentioned earlier however, event sequences appear to be more common in interactive applications. We wouldn't necessarily expect to see the same proportion of the Linux kernel vulnerabilities be caused by event sequences.

The vulnerabilities were analyzed and sorted according to their interaction strength, i.e., how many events contributed to fault propagation. Originally, we expected that most vulnerabilities would be caused by a higher number of events with fewer caused by a small number of events. This hypothesis is opposite to the distribution found in configuration / input testing (Figure 2). The idea being that most bugs caused by a single event would be found early on in the software development process, since it is likely to turn up in testing without any specialized test suites targeted towards events. However, vulnerabilities caused by a much more complex sequence of events would be less likely to be discovered during testing, and therefore would remain in the code during release. The results of the vulnerability analysis are shown in Figure 10 below.

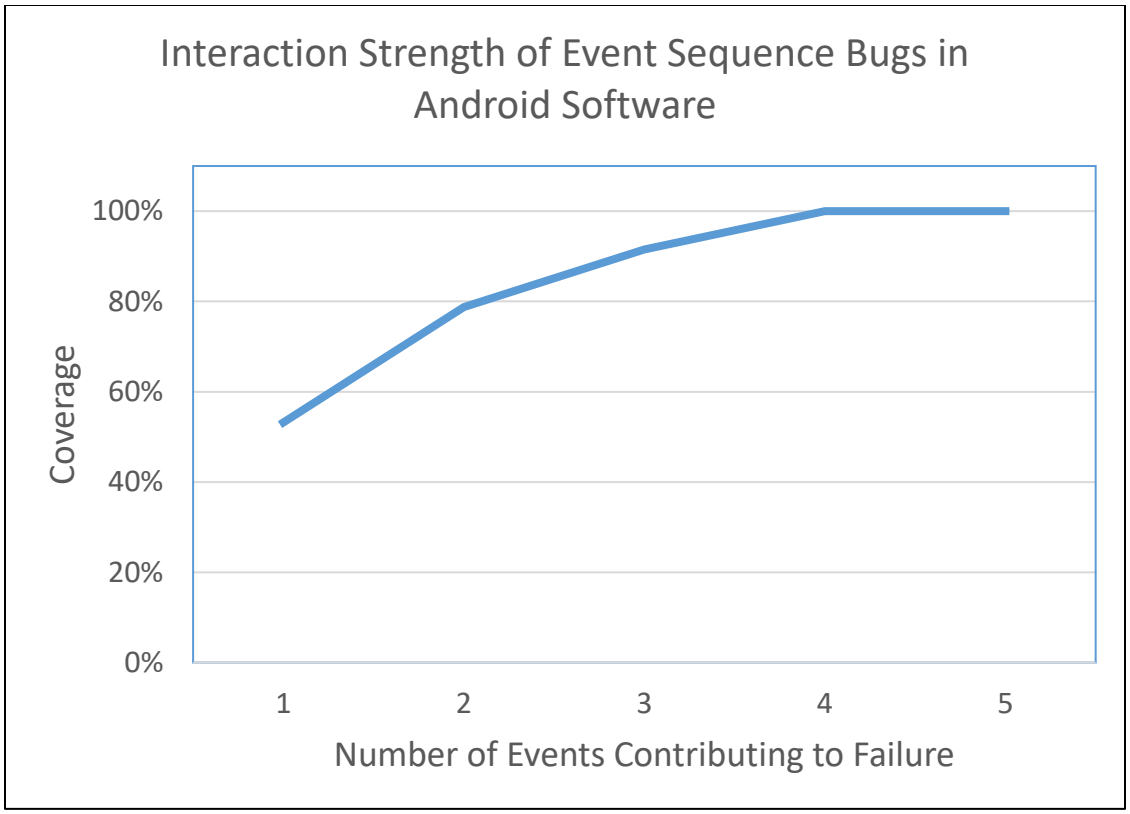

Figure 10. Interaction Strength of Event Sequence Bugs In Android Software

As illustrated in the figure above, the distribution follows that of previous bug analysis studies. Contrary to our original hypothesis, the majority of event sequence bugs were caused by a single event, with progressively fewer bugs caused by two or more events. This is in direct accordance with the interaction rule that makes combinatorial testing so effective. So what is likely the cause for so many event sequence bugs being caused by single events? We believe it could be due to a number of factors, but the main one being a change in software specifications. We will further support this reasoning in the next two sections, however, it should suffice to say that as operating systems, APIs, etc. all evolve, it can result in unintended security holes created in legacy code. For instance, consider an Android application that was designed and developed on Android 5.0, originally meeting all the specifications to run free of issues. After one year the application is no longer in active development, and the Android OS has been upgraded to version

6.0. With the upgrade to Android, new system events are introduced that were not accounted for by the developers of the 3rd party application (as these events were not in the system specifications). This change can result in unintended events being performed that were never actually tested for during development.

A higher number of single event vulnerabilities could also be caused by the lack in understanding of some external software component. An excellent example of this is the Setup Wizard vulnerability CVE-2015-6643, mentioned in chapter 3. The development team working on the Setup Wizard were likely unaware that the Google keyboard settings could be accessed by holding down the "@" symbol, which led to an unaccounted event in the setup wizard software. These single events are never tested during the software development phase since they simply are not known about at the time of testing.

The implications of the distribution obtained highlight the effectiveness that sequence covering arrays can have in detecting these types of bugs. With a correct system model, all vulnerabilities analyzed in this research could be discovered via a 4-way sequence covering array. Additionally, a high number of the vulnerabilities (~90%) could be discovered with a 3-way sequence covering array, which requires an even smaller number of test cases. We would expect that these combinatorial testing methods are used more frequently during testing since pseudo exhaustive coverage of sequences can be achieved with such a small number of tests.

Bypassing the Microsoft Next Lock Screen

The Microsoft Next Lock Screen was tested using the system model in chapter 3. A 4-way sequence covering array was generated based on events $a,\ldots,k$, using the NIST sequence covering array generator tool. From the previous section, we see that a sequence covering array of this size will provide sufficient confidence in finding an event sequence bug, if one exists.

The generated sequence covering array contained 72 tests, so running through each would take very little effort for a small software development team. Of the 72 tests that were run, one critical event sequence bug was discovered. This particular event sequence bug would eventually allow for the complete bypass of the PIN code mechanism, compromising the security of the entire system. Although a 4-way sequence covering array was used, the event sequence bug only had an interaction strength of 2. Therefore, a 2-way sequence covering array (which contains only two tests) could have been used instead.

The security flaw occurred after the two events *openCamera* and *enterPin*. Later we will see that the second event can be substituted with any event that transitions back to the home screen via the pushing of the "home" button. With any screen lock mechanism in place e.g. PIN or pattern, the lock screen could be bypassed by simply opening the camera (either with the soft camera button, or a dedicated hardware camera button), and then transitioning back to enter a PIN code. It should be apparent by our system model that all of our attacks require physical access to the target device. From a hacker's standpoint this may be less attractive, however, this still remains a serious security flaw. Credit card numbers, private messages, photos, and other sensitive information are often stored on mobile phones with the only layer of security being a lock screen.

The following steps were used to trigger the event sequence bug which eventually lead to the complete bypass of the Microsoft Next Lock Screen. This event sequence bug is present on all devices and versions of Android that were used in this research (Sony Xperia Z1, Google Nexus 6P, and Sony Xperia XZ Premium).

1. Open the camera
2. Press the “home” button to transition back to the lock screen to *enterPin*
3. FAILURE

At step 3, the failure occurs for a split second and can easily be missed. When pressing the home button to transition back to the lock screen from the camera application, the home screen flashes briefly before exiting back out. From a hacker’s point of view, this is a foot in the door. At this moment, the goal is seeing what can be achieved in this small window of opportunity.

The eventual bypass of the lock screen varied depending on what device is being used. For example, on the Sony Xperia Z1 running Android 5.1, the attacker can simply open any application on the home screen of the device as it flashes briefly. Whichever application is opened, the attacker is essentially sandboxed in – attempting to leave the application will result in being kicked back out to the lock screen. However, if the attacker chooses the application to open carefully, the lock screen can be uninstalled altogether. For instance, if the Google Play Store icon is on the home screen (it usually is by default), then this app can be leveraged to uninstall the lock screen application via the “uninstall” option within the store.

On the Sony Xperia XZ Premium running Android 8.0, a different method was needed to bypass the lock screen. Attempting to open any application when the home screen flashed resulted in being immediately kicked back out to the lock screen. The Google Voice Assistant was eventually used as an alternative. When the home screen flashes briefly, if the Google microphone icon displayed (again, it usually is by default), then the attacker can quickly activate the voice assistant by clicking it. At this point, the attacker is kicked back out to the lock screen

with the Google Voice Assistant activated in the background. From here the following steps are used:

1. Make a call via “Call [contact]” voice command
2. Press the “Settings” icon in the notification pull down of the call dialog
3. Uninstall Microsoft Next from Settings

In both methods, uninstalling the Microsoft Next Lock Screen was achieved by the 2-way event sequence bug that caused the home screen to flash briefly. Some additional steps were then taken to achieve the actual exploitation of the vulnerability. A few important caveats are of interest to us. First, the second event that allowed us to find the event sequence bug can be substituted by other events that have similar side effects. This goes back to our lazy implementation of the system model. By only including independent events and not including state transitions, many of the events cause the same transitions to be executed. For instance, *enterPin* and *changeWallpaper* both require transitioning to the lock screen’s main interface. So the sequence *openCamera, …, changeWallpaper* would have also triggered the event sequence bug. Second, notice that event *c* and *d* both activate Google Assistant from the lock screen. For our event sequence testing, the events *c* and *d* never actually triggered a bug and the lock screen actually prevented the assistant from launching. Only after triggering the event sequence bug that caused the home screen to flash, did the Google Assistant activate. Finally, there are actually a few bugs that are caused by other sequences of events, that an SCA would not have likely caught. For instance, after discovering that the transition from the phone call to the settings screen in the above exploit, another event sequence bug was realized. If the attacker answers any incoming phone call from the device, then a simple click of the settings button from the phone dialog will bypass the lock screen. However, the SCA would likely not have discovered this

vulnerability due to the fact that the events need be adjacent i.e. *acceptPhoneCall*, *pressSettingsIcon*.

Bypassing the Super Locker Lock Screen

The Super Locker lock screen was tested using a 4-way sequence covering array, generated after the model described in chapter 3. Although the Super Locker lock screen is in active development and is the more popular choice among lock screen applications, we were surprised to discover it contained more event sequence bugs than the Microsoft Next application. Two critical event sequence bugs were discovered during testing, each of which leads to an eventual bypass of the lock screen.

The first event sequence bug occurs after a single event, *F* (click Advertisement). The advertisements that appear in the scrollable window are consistently linked to the Google Play Store. Upon clicking one of these advertisements, the Google Play Store application opens without any prompt for authentication. If the user attempts to leave the Google Play Store, via the home button or some other method, they are kicked back out to the lock screen. However, as shown in the previous section, the Super Locker application can simply be uninstalled once the Google Play Store is opened, therefore completely bypassing the lock screen. This vulnerability was present on the Sony Xperia Z1 (Android 5.1), Google Nexus 6P (Android 7.0), as well as the Sony Xperia XZ Premium (Android 8.0).

The second event sequence bug was discovered on both Sony devices which are equipped with hardware camera buttons. Bug propagation occurs by means of events *I* and *B* (emergency call and open camera via hardware button).

1. Open emergency call screen
2. Push hardware camera button

3. Take a picture
4. Share the picture via Gmail app

Although this may appear be a 4-way event sequence bug, this is not the case. The fault actually occurs on step 2 but is not visible to the user until step 4. First, the application passes control to the emergency call activity which is not managed by Super Locker. At step 2, the camera is launched via the hardware button. Verification that the fault occurs at this step, can be achieved by launching the camera application in any other way – attempting to take and share a picture is not permitted (the Super Locker application intercepts the "share picture" action and prompts the user for a passcode). However, when launching the camera from the emergency call screen, Super Locker is not able to intercept the "share picture" action. We believe this is due to the fact that the camera is launched with a different context in this scenario. The Super Locker developers likely did not consider the fact that the camera activity could be reached from the emergency call screen if the device is equipped with a hardware camera button. Once the Gmail app is launched, the Google Play Store can be reached via a chain of actions within the application. In the previous exploit examples, we mentioned that once inside the Google Play Store, an attacker can uninstall arbitrary user applications from the device.

We should turn the discussion towards whether or not a 2-way sequence covering array would be sufficient for bug discovery in cases such as these. Would a 2-way sequence covering array even be effective at triggering the fault? In our case the 4-way SCA was enough for triggering the fault, however, this is not always the case. Notice that for the bug to occur, the events *openEmergencyCallScreen* and *pushHardwareCameraButton* are required to be adjacent. In the definition of sequence covering arrays, *t*-way event sequences are not required to be adjacent, therefore there is no guarantee that this bug would be discovered. Furthermore, it is

unlikely that the developers would discover this bug with an SCA even if the events weren't required to be adjacent. As mentioned before, the hardware camera buttons were probably not considered while designing the application, therefore it is unlikely that this event would be included in their test suite. Even in our situation, we were not immediately aware that a fault had occurred. However, during each test we would execute other trivial actions (outside the scope of our test suite) to gauge whether or not something changed. For instance, the action *sharePicture* was not included in our test model, but we would often execute the action during each test to see if it still behaved the same. The main reason for not including this action in the system model itself, was that it is primarily related to the camera application (which is accessible from Super Locker), but not directly a part of Super Locker. In this scenario, including *sharePicture* in our system model would have increased our chances of finding the vulnerability.

Performance of the CSCM Tool

The CSCM (Combinatorial Sequence Coverage Measurement) tool is the first tool providing *t*-way sequence coverage measurement that we are aware of. Evaluating the performance of CSCM should provide a good benchmark for future tools to improve upon as well as gauge the practicality of using this tool in practice.

CSCM was developed to run on a single core, as well as with the option to run in parallel. Counting the subsequences of every test is a very natural parallel process so it makes sense to perform those computations on multiple cores. In Figures 11 and 12, the runtime performance between a single core and parallel execution on 7 cores is compared. Note that in Figure 11, the runtime is measured in seconds, while Figure 12 measures runtime in minutes. Both sequence covering arrays were generated over 26 events, with the 3-way SCA containing 26 tests, and the 4-way SCA containing 144 tests. The system models in chapter 3 contained less than half the

number of events used in this benchmark, so it is expected that this example is much more computationally expensive than many real-world scenarios might call for.

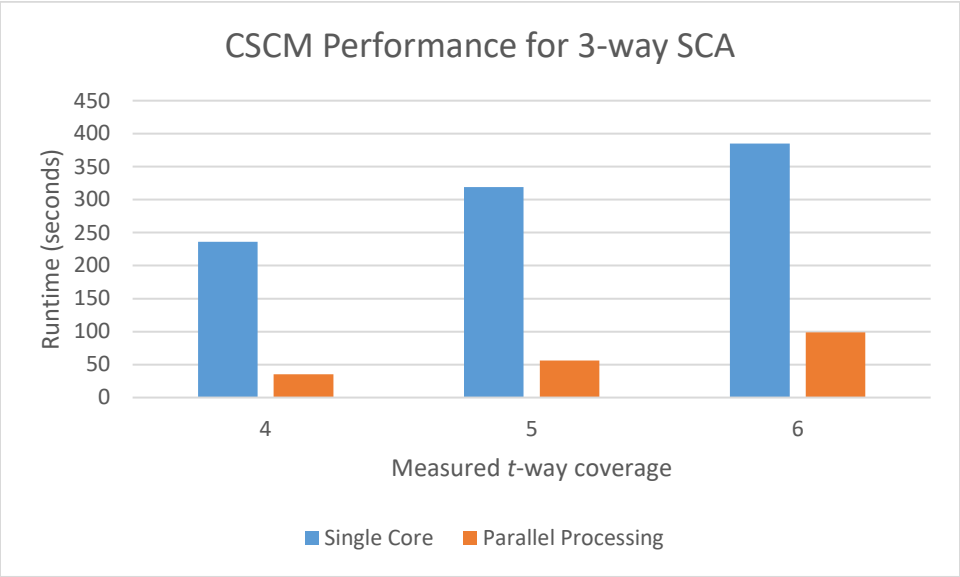

Figure 11. CSCM Performance on 3-way SCA

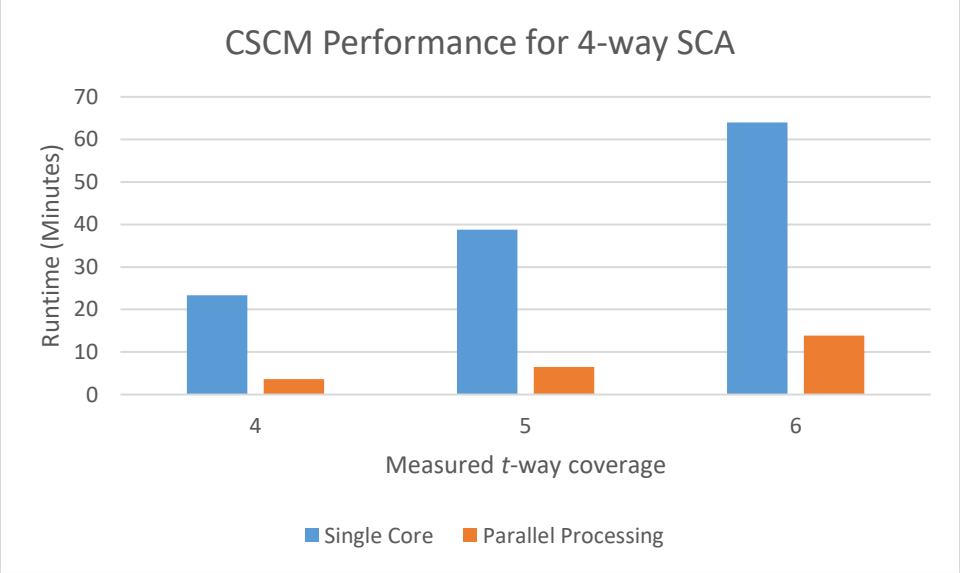

Figure 12. CSCM Performance on 4-way SCA

The CSCM tool is freely available for download via the link in appendix A. In the future, the tool will continue to be developed to support other features necessary in real world testing environments. For example, the ability to include constraints would be beneficial for system models with restrictions on the order of events. For instance, if event *a* can not be executed before event *b*, then this *t*-way sequence should not be counted during coverage measurement.

Implications

The distribution of interaction strengths for *t*-way event sequence bugs provides valuable insight into the practicality of discovering these bugs. Out of 592 vulnerability reports, 7.9% of the vulnerabilities were caused by some sequence of events. Furthermore, each of these vulnerabilities could be detected via a 4-way sequence covering array. In practice, this highlights the time and money that can be saved while testing for bugs of this type. Pseudo-exhaustive coverage of event sequences can be achieved in very few tests, leaving developers more time to test for the other, more common types of bugs. Sequence covering arrays were also very effective in discovering bugs in our black-box scenarios, as we were able to find several critical vulnerabilities in the two popular 3rd party lock screen applications we tested. If software developers continue to leave event sequence bugs in products they release, hackers can effectively use these combinatorial methods to exploit the software.

CHAPTER V

CONCLUSION & FUTURE WORK

This chapter will first summarize the major results of this thesis, highlighting the contributions we have made to the cybersecurity field as well as to combinatorial software testing research. Then we will discuss the future direction of this work.

Conclusion

In this research we explored the effectiveness of using sequence covering arrays in testing mobile applications, specifically in black-box environments where little to none is known about the internal structure of the system. Sequence covering arrays have been hypothesized to be useful in testing interactive systems, but until now, the practicality of such methods was not heavily studied.

First, the distribution of interaction strengths for event sequence bugs was determined via vulnerability analysis. The distribution was similar to that of software bugs studied in previous research, in that the interaction rule still holds. The majority of event sequence bugs analyzed were caused by one or two events, with progressively fewer bugs caused by 3 or 4 events. The implications of this finding are that 4-way sequence covering arrays can provide pseudo-exhaustive testing of event sequences. However, more event sequence bugs should be analyzed in future research to further evaluate this finding. Event sequence bugs appear to be much less common than ordinary software bugs e.g. those caused by incorrect bounds checking, overflows, etc. In our case, only 7.9% of the vulnerabilities analyzed were classified to have been caused by event sequences. For this reason, much of our research involved analyzing hundreds of unrelated software bugs.

Next, we utilized sequence covering arrays in a real world black-box testing scenario. Two popular 3rd party lock screen applications on the Android Operating System were tested for vulnerabilities. In both cases, the source code was unavailable, and our test suite was generated based on external observations of the system. While testing the first application, Microsoft's Next Lock Screen app, we discovered a vulnerability that was triggered with a 2-way sequence covering array. Exploitation of this vulnerability allowed us to completely bypass the lock screen mechanism, compromising the security of the entire system. Similar results occurred while testing the second application, Super Locker lock screen. Using sequence covering arrays we were able to find 4 different critical vulnerabilities, each of which led to the eventual bypass of the lock screen. The implications of these experiments illustrate that sequence covering arrays can be highly effective in triggering critical software bugs in real world scenarios. Furthermore, this increases the incentive for software testers to use these combinatorial testing methods before releasing the final version of their product. If hackers can effectively use these methods to find software bugs without internal knowledge of the source code, then it is essential that these types of software bugs be removed early on in the software development phase.

Lastly, a new software tool was developed for measuring the combinatorial coverage of events in test suites. CSCM (Combinatorial Sequence Coverage Measurement) was developed using the Haskell programming language and is freely available for download. We believe this tool will be useful for measuring the combinatorial coverage of events in existing test suites, helping software testers gauge the robustness of their current test setup. For instance, if a testing team uses CSCM on their current test suite and deems it to not have adequate coverage, they may choose to switch to a sequence covering array. We also believe that this tool will be beneficial for comparing various sequence covering array generation algorithms. If two SCA generators are

equally fast in producing similar sized covering arrays, CSCM can determine which algorithm yields a higher (t+1)-way coverage.

Future Work

The basis for combinatorial software testing rests on the interaction rule i.e. the fact that most software bugs appear to be triggered by 6 or fewer inputs or configuration parameters. The distribution acquired from this research points to the same phenomenon for event sequence bugs. In both cases, more analysis of software bugs will be beneficial. We believe that the future direction of this research involves further analysis of event sequence bugs. However, since dedicated databases of these types of bugs do not exist, this work is very time consuming. Researchers will spend a lot of time analyzing bug reports that are not related to event sequence bugs. In our case, we analyzed hundreds of vulnerability reports, however, only 49 were eventually classified to be caused by event sequences. There is currently work being done in improving vulnerability databases such as the CWE (Common Weakness Enumeration) to a more standard "periodic table of bugs [14]." We believe that including event sequence bugs in these efforts would be beneficial to the security community.

Second, we believe that the developed CSCM tool would be useful in comparing existing sequence covering array generation tools. In previous research, many sequence covering array algorithms have been proposed [11] [15]. One effective method of comparing algorithms would be measuring the (t+k)-way coverage that each provides on identical system models. For instance, a generated 3-way SCA that also provides 80% 4-way coverage would be more efficient than a 3-way SCA that only provides 40% 4-way coverage.

Lastly, we have further interest in using sequence covering arrays in practice. For this research, only two 3rd party applications were tested, both of which were lock screens. Using

these methods on a more diverse set of software applications would be interesting work for both the software testing and hacker communities.

REFERENCES

[1] Homaei, Hossein, and Hamid Reza Shahriari. "Seven Years of Software Vulnerabilities: The Ebb and Flow." *IEEE Security & Privacy 15.1* (2017): 58-65.

[2] D.R. Kuhn, D.R. Wallace, A.M. Gallo, Jr., Software Fault Interactions and Implications for Software Testing, *IEEE Transactions on Software Engineering*, vol. 30, no. 6, June 2004, pp. 418-421.

[3] Hagar, J. D., Wissink, T. L., Kuhn, D. R., & Kacker, R. N. (2015). Introducing combinatorial testing in a large organization. *Computer, 48*(4), 64-72.

[4] Kuhn, D. R., Higdon, J. M., Lawrence, J. F., Kacker, R. N., & Lei, Y. (2012, April). Combinatorial Methods for event sequence testing. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on* (pp.601-609). IEEE.

[5] Hartman, Alan. "Software and hardware testing using combinatorial covering suites." *Graph theory, combinatorics and algorithms*. Springer, Boston, MA, 2005. 237-266.

[6] Ratliff, Z. B., Kuhn, D. R., Kacker, R. N., Lei, Y., & Trivedi, K. S. (2016, October). The Relationship between Software Bug Type and Number of Factors Involved in Failures. In *Software Reliability Engineering Workshops (ISSREW), 2016 IEEE International Symposium on* (pp. 119-124). IEEE.

[7] Android Activity Lifecycle. Android Developers. Web. https://developer.android.com/guide/components/activities/activity-lifecycle.html [Last Accessed on April 2018, 8]

[8] AOSP. Android Security Bulletin. Web. https://source.android.com/security/bulletin/ [Last Accessed on March 2018, 4]

[9] NIST. National Vulnerability Database. Web. http://nvd.nist.gov [Last Accessed on March 2018, 4]

[10] CVE. Common Vulnerabilities and Exposures (CVE) Numbering Authority (CAN) Rules. Web. http://cve.mitre.org/cve/cna/CNA_Rules_v1.1.pdf [Last Accessed on March 2018, 4]

[11] Yu, L., Lei, Y., Kacker, R. N., Kuhn, D. R., & Lawrence, J. (2012, July). Efficient algorithms for t-way test sequence generation. In *Engineering of Complex Computer Systems (ICECCS), 2012 17th International Conference on* (pp. 220-229). IEEE.

[12] Microsoft Next Lock Screen. Microsoft Garage. Web. https://www.microsoft.com/en-us/garage/profiles/next-lock-screen/ [Last Accessed on March 2018, 16]

[13] Super Locker Lock Screen. Augeapps. Web. https://play.google.com/store/apps/details?id=com.augeapps.locker [Last Accessed on March 2018, 20]

[14] Bojanova, I., Black, P. E., Yesha, Y., & Wu, Y. (2016, August). The Bugs Framework (BF): A Structured Approach to Express Bugs. In *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on* (pp. 175-182). IEEE.

[15] Ahmad, M. Z. Z., Othman, R. R., & Ali, M. S. A. R. (2016). Sequence Covering Array Generator (SCAT) For Sequence Based Combinatorial Testing. *International Journal of Applied Engineering Research*, *11*(8), 5984-5991.

APPENDIX

COMBINATORIAL SEQUENCE COVERAGE MEASUREMENT

This software facilitates measuring the combinatorial coverage of event sequence test suites. The software is freely available online and includes a user manual along with several of the sequence covering arrays used in this research for evaluating the performance of the tool.

The notion of "coverage" refers to the percentage of *t*-way permutations of events that are included in the test suite. A test suite that has 100% *t*-way coverage will include all *t*-way permutations of events. Furthermore, a test suite that has 100% *t*-way coverage will also have 100% (*t*-1)-way coverage.

Available on GitHub at https://github.com/zachratliff22/CSCM-Tool.git