Layout Investigation of GUI Navigational Interactions of Android Applications

by

JOYLYN ALEXANDER LEWIS
B.E., Computer Engineering, Sardar Patel College of Engineering, India, 2008

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2019

Chicago, Illinois

Defense Committee:
Dr. Mark Grechanik, Chair and Advisor
Dr. Balajee Vamanan
Dr. Luís Gabriel Ganchinho De Pina

ACKNOWLEDGMENTS

I am sincerely grateful to my advisor Dr. Mark Grechanik for his patient support and encouragement throughout the duration of my thesis. I want to thank him for trusting me on my capabilities and for giving me the opportunity to work as a Graduate Research Assistant under his guidance. His advice, inputs on crucial topics have helped me in conducting the required research for my Master's thesis. This would not have been possible without his guidance and support.

I also want to sincerely thank Dr. Balajee Vamanan and Dr. Luís Pina for serving as members of the defense committee.

JAL

CONTRIBUTION OF AUTHORS

The content in the sections: *1.1 Overview of App GUI Framework* and *1.2 Overview of GAP State Machine* of Chapter 1 has been referenced from Dr. Mark Grechanik's previously published work [1–8]. The section 1.1 Overview of App GUI Framework provides the basic framework of an application's interface and provides a base for my thesis work. The section 1.2 Overview of GAP State Machine describes the representation of an application as a finite state machine and helps me to elaborate on what my thesis aims to achieve. Dr. Mark Grechanik has authorized me to reference and include the materials from his published work for my thesis.

The content in Chapter 2 that includes the sections: *2.1 Attack by exploiting Assistive Technologies* and *2.2 SEAPHISH: Defense by deception* has been referenced from Dr. Grechanik's previous unpublished NSF proposal. Dr. Grechanik has authorized me to use all the materials from his NSF proposal for my thesis. The Chapter 2 on Motivation introduces Accessibility Services and how it is required by users with disabilities. The section 2.1 Attack by exploiting Assistive Technologies describes the attacker model and the section 2.2 SEAPHISH: Defense by deception describes a platform aimed towards protecting against such an attack.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACTEve	Automated Concolic Testing of Event-driven programs
APK	Android Package Kit
App	Application
AS	Accessibility Services
A^3E	Automatic Android App Explorer
DbD	Defense by Deception
GAP	Graphical User Interface-based APplication
GATOR	proGram Analysis Toolkit fOr andRoid
GUI	Graphical User Interface
GUID	Globally Unique Identifier
MA^2P	Mobile Assistive APplications
SDK	Software Development Kit
SEAPHISH	SEcuring Accessibility using PHISHing
UWD	Users With Disability
WTG	Window Transition Graph

SUMMARY

Nowadays, *Graphical User Interface (GUI)-based APplications (GAPs)* are universal for personal and business needs. Hence, making GAPs accessible to users with disabilities is extremely important. Unfortunately, about 50 million people have disabilities in the United States itself and over 600 million worldwide [11–14]. Thus, users with disabilities should be protected from possible malware attacks.

SEAPHISH (SEcuring Accessibility using PHISHing) is a platform designed towards protecting against an attack by providing defense by deception. A simulation for SEAPHISH can help govern the circumstances when an attack against a specific application can be done with a high degree of probability. But performing effective simulations requires a fundamental understanding of the properties of GUI layouts of applications (apps) at large.

This work focuses on providing a framework that analyzes GUI layouts and their transitions using a large base of approximately three million Android apps. We discuss various state-of-the-art tools that use different strategies in traversing GUI layouts. Although each tool has its own unique features, we build our solution using the tools that best help in building a GUI model of an application and we run it on a small base of 200+ Android apps.

We hope the investigation done in this thesis enhances SEAPHISH with statistically significant real-world constraints thereby providing defense against malware and reducing the security vulnerabilities faced by users with disabilities.

CHAPTER 1

INTRODUCTION

The content in sections *1.1 Overview of App GUI Framework* and *1.2 Overview of GAP State Machine* has been referenced from Dr. Mark Grechanik's previous published work: *"Grechanik, M., Mao, C. W., Baisal, A., Rosenblum, D., and Hossain, B. M. M.: Differencing graphical user interfaces. In 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), pages 203–214, July 2018"* [1–8].

A mobile application (app) is a software program or a computer application written to operate on a mobile device. The mobile app market is the fastest growing sector in the mobile industry [15]. Mobile apps use multi-touch screens, have access to the location of the device and these apps differ from desktop apps which operate on computers and web applications that operate on web browsers instead of directly operating on a mobile device [16].

There are different operating systems and platforms like Android (Google), iOS (Apple), Windows (Microsoft), Blackberry (Research in Motion) that allow mobile devices to run apps and other software programs. Android and iOS are two of the major mobile operating systems that consume a collective 98.7% of the market share in the USA [17]. However, Android has a somewhat larger customer base which is mainly due to its compatibility with different manufacturers like HTC, Motorola, LG, Samsung etc whereas iOS only operates on Apple products. As reported in August 2019 [18], the Android operating system holds a 76.23% market share worldwide.

The mobile apps can be downloaded from an app store or app marketplace. Each of the mobile operating systems (like Android, iOS, Windows etc) has its own store for distributing and making apps available for downloads. For example, Google Play Store is the official app store for Android where users can search and download apps that are developed with the Android software development kit (SDK) and published through Google [19–21].

In this chapter, we look into basics of the GUI framework in section 1.1 Overview of App GUI Framework. We next look at how an app can be represented as a state machine in section 1.2 Overview of GAP State Machine. We then look into basics of Android GUI components in section 1.3 Overview of Android GUI components. We conclude this chapter by providing a high-level description of the problem that this thesis aims to solve along with our contributions.

1.1 Overview of App GUI Framework

A GUI framework consists of an extensible and reusable set of GUI objects with well-defined interfaces that can be specialized to produce and run custom applications [22]. Figure 1 shows a basic model of GUI frameworks [1].

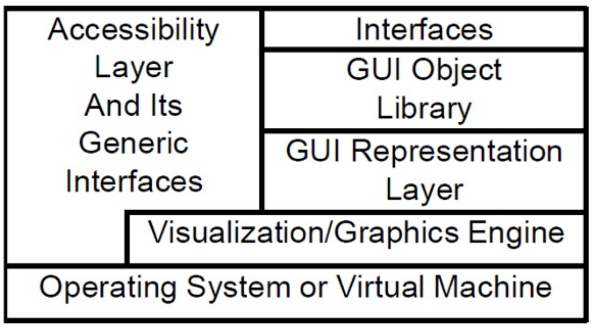

Figure 1: A model of GUI Frameworks from [1]

As shown in Figure 1, a GUI framework [1] typically consists of four main components: GUI representation layer, GUI object library, GUI interface, and the accessibility layer with its generic interfaces. The GUI representation layer defines how the GUI objects are programmatically represented as data structures in computer memory. As an example, HTML web pages are represented using a document object model in web browsers while Windows defines proprietary data structures for GUIs and the events that they send and receive. The visualization/graphics engine interprets these data structures and visualizes them using some predefined settings for styles and layouts [23, 24].

To allow users to interact with GUI objects, the underlying operating system or virtual machine provides queues for receiving user inputs from peripheral devices (e.g., mouse, keyboard or a touch screen) and translates these inputs into event data structures that are passed to the corresponding GUI objects using its interfaces. GUI object libraries contain implementations of GUI objects and expose their interfaces; these libraries are extensible and many third-party vendors offer implementations of sophisticated GUI objects for different GUI frameworks. In general, GUI frameworks, which are developed by different vendors, expose diverse interfaces.

1.2 Overview of GAP State Machine

Figure 2 presents a representation of a GAP as a state machine. In this figure, nodes are the tree representations of its GUIs and transitions are labeled with actions on certain GUI objects that trigger corresponding method calls within the GAP. Thus, the GAP starts with the main window consisting of a tree representation of the GUI objects and when an action is taken against a GUI object like Select, the state of the GAP changes leading to the next

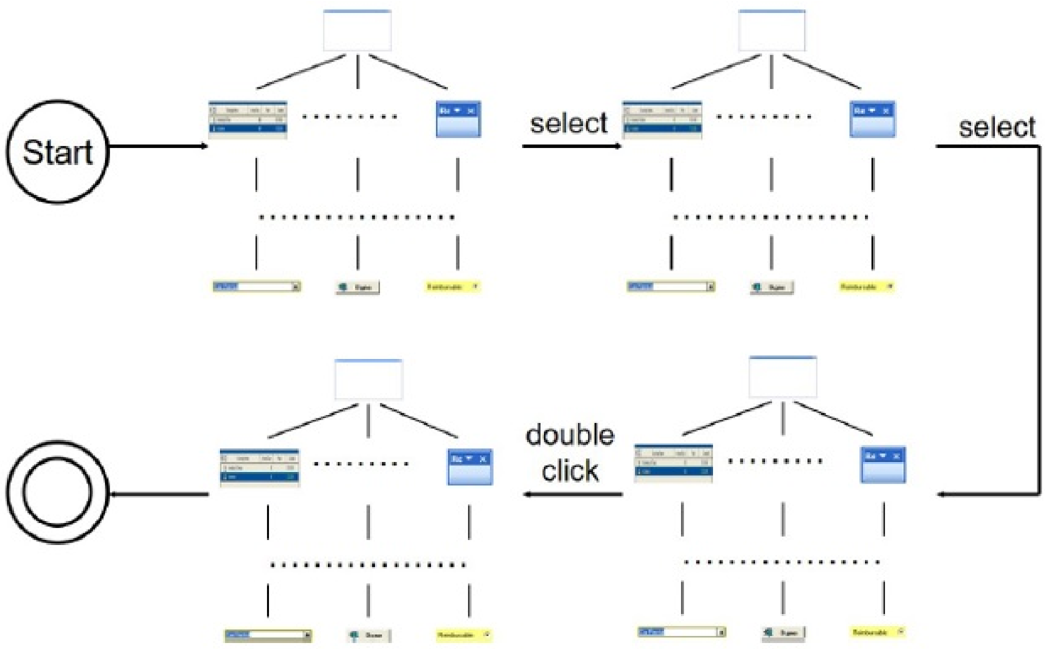

Figure 2: Representation of a GAP as a state machine from [1]

node, which is another tree representation and so on. In event-based systems like Windows or Android, each GAP has a main window, which is associated with the event processing loop. Closing this window causes an app to exit by sending the corresponding event to the loop. The main window contains other GUI objects of the GAP. A GAP can be represented as a tree, where nodes are GUI objects and edges specify that children objects are contained inside their parents. The root of the tree is the main window, the nodes are container objects, and the leaves of the tree are basic objects [2–8, 25].

Each GUI object is assigned a category (class) that describes its functionality. For example, in Windows, the basic class of all GUI objects is the class *Window*. Some GUI objects serve as containers for other objects, for example, dialog windows, while basic objects (e.g., buttons and edit boxes) cannot contain other objects and are designed to perform some basic functions. Thus, different GUI trees include their topologies and labels of their nodes [1].

1.3 Overview of Android GUI components

An Android app that consists of different user interface components is built using the basic objects: View and ViewGroup [9,26]. View is the base class for widgets, which are used to build interactive user interface elements like buttons, textboxes etc [27]. A ViewGroup can contain other views that can be referred to as children. Thus, the ViewGroup acts as the base class for other view containers and layouts [28]. It can be referred to as an object that holds other View or ViewGroup objects in order to express the layout of a user interface.

A set of subclasses of both View and ViewGroup class are defined in the Android framework that provides various input controls and different layout models like relative or linear layout. Figure 3 depicts a simple hierarchy of the View and ViewGroup objects.

There are two ways to declare the layout for an app, either by instantiating View objects in code and a tree can be built or by defining the layout in an XML file. For a view, the name of an XML element is the Android class it represents. So an XML element <TextView> generates a TextView widget in the user interface, and the XML element <LinearLayout> generates a LinearLayout ViewGroup. Figure 4 shows a simple vertical linear layout with a TextView and a button. On loading the resource layout in the app, each node of the layout is initialized by Android into a runtime object that can be used to query the state of the object or to define additional behaviors or to modify the layout [9].

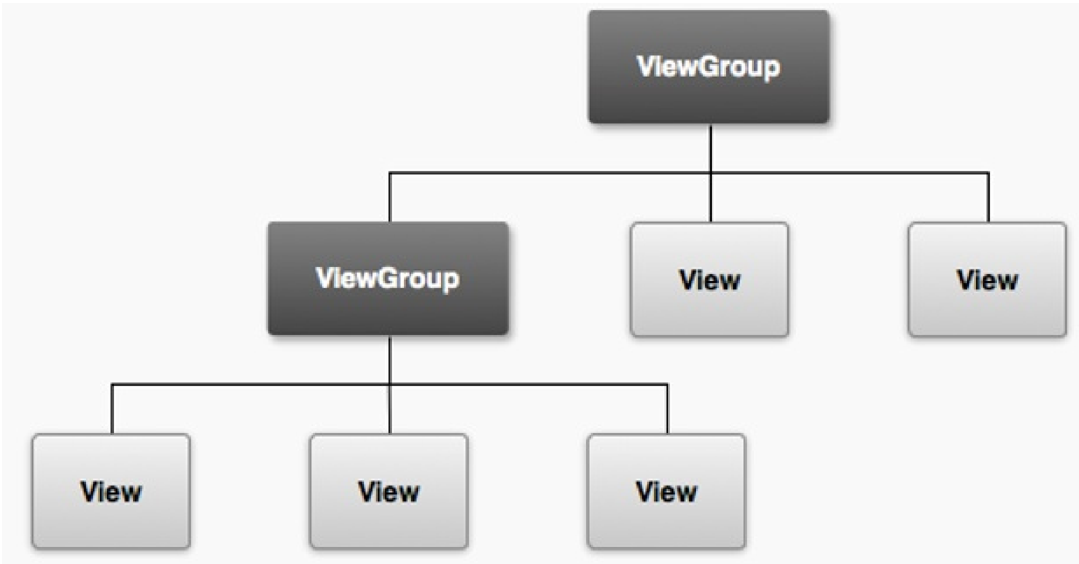

Figure 3: A view hierarchy, which defines a UI layout from [9]

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <TextView android:id="@+id/textExample"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Example of a TextView" />

    <Button android:id="@+id/buttonExample"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Example of a Button" />
</LinearLayout>
```

Figure 4: Declaring XML layout for a textview and button

The fundamental building blocks of an Android app are the app components [29], each of which could be an entry point through which one could enter the app. There are four basic types of app components:

1. **Activity:**

   An Activity denotes a single screen with a user interface and it is the entry point through which a user can start interacting with the app. For example, the Messages app consists of multiple activities like one activity that displays list of messages, another activity that can compose a new message and another to edit a message. Each activity may be independent of the others but they function together to provide a cohesive user experience.

2. **Service:**

   A service is an entry point component that runs in the background without providing a user interface. It can perform long running processes or tasks for remote operations. An example of a service is the one in radio that allows a user to listen to music in the background although the screen maybe locked or another app may have been opened. An activity can initiate a service and let it run or bind to it so that it can interact with it [29].

3. **Broadcast Receivers:**

   Broadcast receivers are components that listen and react to events. These components use intent filters to select events of interest. An intent [30] is a messaging object used to request an action from another app component. An intent filter specifies the type of intents that can be accepted based on certain attributes of an intent like its action, data and category. Thus, an app can register for which broadcast events it would like to be notified of. For example, an app may register to be notified of a broadcast event like the screen has turned off or the battery is low.

4. **Content Providers:**

   A content provider is a component that manages a shared collection of application data that may exist in a file system, or in a SQLite database, or on any other persistent storage location that the app can access. Thus, other apps that have appropriate access permissions to the content provider can access, query or even modify the data. For example, a content provider can manage the user's contact information.

In our thesis of exploring GUI layouts, out of the four basic app components, we mainly focus on Activities as only Activities consist of GUI screens and objects. The other three components do not provide any user interface.

Android Package Kit (APK) is the package file format for the distribution and installation of Android apps. An APK file is built by first compiling a program for Android followed by packaging all of its parts into one container file [31]. This APK file consists of program code (that includes .dex files i.e. the Dalvik Executable files generated on compiling Android programs), resources, assets, certificates and the manifest file. Resources consist of static content files that the code uses such as GUI layout declarations, user interface strings and other instructions. Assets help with including raw data files like text, xml, fonts etc. for the app. The certificates are required by Android for digitally signing the apk before the apk file is installed on a device or updated. The manifest file i.e. AndroidManifest.xml [29] file is where all the required components, features and permissions for running an Android app are declared.

Similar to other file formats, APK files can have any name, provided that the file name ends with the extension ".apk".

Apktool is a tool for reverse engineering third party, binary, closed Android apps [32]. Using the decode option on Apktool, we can run the command **"apktool d sampleApk.apk"** where **d** represents decode option and **sampleApk.apk** represents any Android apk file. Running this command helps in extracting all the resource files for the app [33].

Thus, having looked into the basics of the Android framework, we provide here a high-level description of the problem and work done in this thesis. Accessing an app on a smartphone involves events like click, touch, scrolling etc. Accessibility technologies aid users with disabilities in accessing an app on a smartphone. However, such technologies are not fully secure. Chapter 2 describes a possible attack when Accessibility services are turned on. We use an idea stated by Dr. Grechanik in his awarded NSF proposal that Defense by Deception can be used to defend against malicious applications that users installed on their smartphones and gave sufficient permissions to access and manipulate smartphone services the same way that the users can. The defense works even if the user allowed the accessibility service permissions to the installed application. However, to further enhance this defense mechanism, we need to obtain properties of GUI layouts and their transitions from a large collection of Android apps.

In this thesis, we explore various state-of-the-art tools that use different strategies in traversing GUI layouts. We created a hybrid framework where the tool Backstage performs static analysis of the layouts to help build a GUI model of the app and for cases where Backstage is not successful, we propose to use AndroidRipper which performs dynamic analysis to generate the GUI models. Using these models, we investigate the layouts of Android apps by collecting statistics on various GUI elements and screens. This investigation enhances the defense mechanism with statistically significant real-world constraints thereby providing defense against malwares and reducing the security vulnerabilities faced by users with disabilities.

CHAPTER 2

MOTIVATION

The content throughout this chapter including the sections *2.1 Attack by exploiting Assistive Technology* and *2.2 SEAPHISH - Defense by deception* has been referenced from Dr. Mark Grechanik's previous unpublished NSF proposal.

The Android framework provides the Accessibility Service (AS) feature that is extended from Android's Service component [34]. This service enhances user interfaces in order to assist users with disabilities (UWD), or others who may for some reason find it difficult to fully interact with a device. In these cases, users may need additional or alternative feedback such as text-to-speech or haptic feedback i.e. the use of touch to communicate with users [35].

Since there are hundreds of disabilities that impair people in vision, movement, thinking, remembering, learning, communicating, and hearing, UWDs need assistance in using GAPs by enhancing their GUIs, [1–4, 36, 37] and Mobile-Assistive APplications (MA^{2}Ps) are designed to provide these enhancement services. There are several hundred MA^{2}Ps on the Google Play store that request permissions for Accessibility Service [38, 39], and two such MA^{2}Ps were downloaded by at least ten million people together and they have a rating of 4.3 out of 5 stars [40, 41]. Although there are hundreds of GUI accessibility approaches [42–51], there is almost no research to provide security guarantees to UWDs [38, 52].

It has been said by many distinguished people including Mahatma Gandhi, Hubert H. Humphrey, and Cardinal Roger Mahony, that the measure of a civilization is how it treats its weakest members. It is a formidable challenge to achieve a measure in which no UWD feels different from other users when working with GAPs to accomplish everyday tasks [53–59]. Since designing accessible GAPs for users with various types of disabilities is very challenging [60–65] and legally required [66,67], a large MA^2P marketplace exists where there is no control over the quality of these applications. Of course, money-stealing applications have long existed [68, 69], however, malware in MA^2Ps is particularly reprehensible as it uses weaknesses of assistive technologies to take advantage of UWDs to steal their financial and other private information.

The following Section 2.1 describes a possible attack against UWDs and Section 2.2 describes SEAPHISH (SEcuring Accessibility using PHISHing), a platform aimed towards protecting against such an attack by providing defense by deception.

2.1 Attack by exploiting Assistive Technology

Accessibility technologies are mandated by the law [67] and a common accessibility architecture is designed for different platforms [70, 71] that can be used to form attacks on UWDs as it is shown in Figure 5. The workflow starts in the upper right corner where the developer of a MA^2P releases it to the app store, and it is eventually downloaded by a UWD to her/his smartphone. The MA^2P developer owns or controls a MA^2P server to which the MA^2P connects and transmits the data, e.g., to offload computationally expensive analyses from smartphones. However, sending the UWD's data to the MA^2P server may lead to security violations. For example, a malicious MA^2P that reads a financial statement to a UWD may send this data out to an external server.

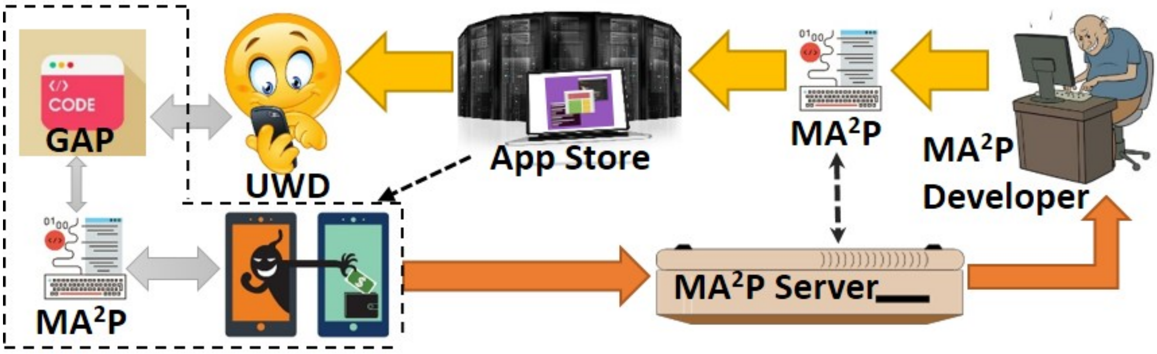

Figure 5: Using assistive technologies in attacks on Users With Disabilities (UWDs)

Accessibility technologies largely bypass permission-based models. A general problem with permission-based security models is that the users do not follow the least privilege principle when granting permissions, and even in the absence of disabilities users cannot make informed decisions about which permissions are really needed. UWDs routinely approve the accessibility permission requested by the MA^{2}P, and it is given access to the accessibility Application Programming Interface (API). In general, operating systems do not properly enforce permissions [72–85]. As a result, GAPs obtain permissions that are unnecessary, and an investigation of 940 applications showed that one-third are over-privileged [73, 80, 86–89]. Moreover, studies showed that users often are not prepared to make informed privacy and security decisions to select from over 130 different permissions [87, 90–95], especially considering high complexity of GUIs [96]. *A key component of the accessibility API calls is to simulate a user of the smartphone, so that programmers can mimic user interactions with GAP by accessing and controlling GUI objects programmatically from MA^{2}Ps. This fundamental strength of assistive technologies* *is also their fundamental security weakness, since it is very difficult to create general permissions that will not affect the usability of MA^2Ps when considering hundreds of different impairments of UWDs* [38].

Moreover, developers find ways to bypass permission checks altogether [97–100]. Interestingly, many GAPs expose private data through GUI objects [101, 102]. An example of this is an edit box for a financial app that displays the user's account balance. Studies with blind smartphone users show they are unaware of or not concerned about security threats [103, 104].

Using the workflow that is shown in Figure 5 we show how MA^2Ps can attack UWDs by stealing sensitive information, which could cause serious harm to the individual, organization or company owning it if this information is compromised through alteration, corruption, loss, misuse, or unauthorized disclosure [105, 106]. Recent investigation of the accessibility technologies uncovered a wide array of security exploits [38], however, here we focus on exploits that steal sensitive information from UWDs.

1. A MA^2P obtains the data from the GUI objects of some GAP, enhances this data for UWDs, and transmits it to the MA^2P server, thus releasing sensitive information in this data into the wild. Interestingly, the MA^2P may not be malicious, it may use the power of server offloading for additional processing of the data. In general, distinguishing clearly between malicious and unwanted behaviours is a very difficult problem [107–110]. Of course, the MA^2P that requests accessibility permissions may not even use the accessibility to provide assistive services, and it can masquerade as an assistive application to steal sensitive information.

2. MA^2P developers use the accessibility API calls to inject threads and register callback functions for different events produced by GUI objects of some GAP. For example, MA^2P called InputObserver utilizes these hooks to measure UWD's usage statistics [111]. In general, such MA^2Ps may be viewed as a case of plagiarizing other smartphone apps [112] or a library modification threat [113]. Thus, an attacking MA^2P can avoid protection mechanisms that analyze it to determine if it tries to send sensitive data outside the smartphone [86]. Hacker tools like this have been used to attack desktop GAPs for a few years [114].

3. A MA^2P can compose others GAPs with access to smartphone on-board sensors to attack UWDs [115, 116]. It is an instance of the documented attack where sensory malware can convey raw sensor data (e.g., video and audio) to a remote server [117]. For example, consider a MA^2P that reads a financial statement to a UWD and it spawns a third party background application on the same smartphone that uses API calls to access the microphone and the phone hardware to capture the voice that is read by the MA^2P and to send it to an external server or even an answering machine. We easily realized this attack using Capture, a display-centric text recorder with real-time access to foreground and background process windows that integrates with the accessibility layer [118]. Of course, attacking sensors on smartphone is not new – PlaceRaider builds rich, three dimensional models of indoor environments for remote burglars [119], but combining different GAPs under the guise of assistive technologies is a new type of attack. Currently, there is no unified approach to protect UWDs from these attacks using the accessibility

API calls while preserving usability, which is extremely important for UWDs [120]. A recent exploratory user study with visually impaired participants revealed serious security concerns that are still not solved [121], and it is not clear how many UWDs are victims of these attacks.

2.2 SEAPHISH - Defense by deception

The idea of using deception is not new to protect computer hardware and software against attack – its main idea is to make the attacker do a lot more work to carry out a successful attack. A well-known use of defense by deception (DbD) strategy is in honeypots [122], which are simulated computer environments where attackers are presented with realistic configurations of the computer systems into which they gained illegal access. To determine if they interact with a real computing environment the attacker must spend time and resources to investigate it, and while doing so it enables the defense mechanisms to detect the attacker and to take countermeasures. Similarly, other deception techniques, strategies, and algorithms are used to increase the probability that the attacker constructs beliefs that lead the attacker to take certain actions, which may require the attacker to invest time and resources. Of course, the beliefs that are instilled by the deception strategy lead the attacker to gamble on an easy large payoff with minimum invested effort. DbD strategy is widely used in military operations where sophisticated deception strategies are used, often with an extensive network of human participants to make military intelligence make wrong conclusions about the battlefield. Unfortunately, it is much more difficult to construct algorithmic DbD strategies to construct and deploy secure computer systems, since deceptions are based on specific scenarios and making general DbD strategies work in concrete cases is very difficult.

This difficulty is partially explained by the element of surprise when deploying deception as a protective weapon. If the attacker knows about the deception, then, depending on the payoff the attacker may find a hole in the DbD strategy and use it to bypass the defense. Regarding honeypots, there are multiple ways for the attacker to determine if the computing environment is real [123]. Of course, as attackers grow more sophisticated so do the defenders and the cat-and-mouse game evolves into finding and exploiting specific holes in the given DbD system before they are patched. A central element of this game is that vulnerabilities and patches in the DbD systems are specific to the system and cannot be applied to other DbD systems without significant modifications if at all.

More importantly, all existing DbD strategies and algorithms protect actual systems from attackers penetrating them – once the attacker managed to subvert the defenses, it is game over and the expensive shutdown and cleanup with damage assessment are in order. These DbD strategies are based on the model where the attacker's goal is to obtain administrative or even read access to the protected system. However, this model is not a good fit for the reality where smartphone users download applications (i.e., mobile apps or simply apps) based on how they may address the needs of the users. Not only many smartphone users do not give serious thoughts about permissions requested by the downloaded app, but also they often gamble with making decisions about downloading and using apps that come from unverified sources. And once these apps are installed on the smartphone with approved permissions, they can perform all actions that can be done by the authorized users of this smartphone. Hence, the existing DbD strategies cannot be applied to this situation.

The other nascent trend is that malicious apps can collaborate over the Internet to distribute computational work in analyzing the environments where DbD is applied. Just like cloud computing is used to parallelize computations by splitting them among millions of computer servers that run instances of the same application (i.e., the map/reduce model), malicious applications can use their installed instances to determine if the DbD strategy is applied. In the case of honeypots, the attacker may run an exploration canary application in many environments and each instance of this canary application will explore a part of the environment sharing the obtained information with other instances to decide if the environments share the same characteristics and thus may be honeypots. Hence, with minimal amount of work the attacker can use the Internet-level parallelism to subvert DbD by obtaining combined information that changes the belief of the attacker about the environment where the attacker plans to commit abuse.

Accessibility Technologies: Since we cannot access and manipulate GUI objects as pure programming objects (they only support user-level interactions), we use accessibility technologies as a universal mechanism that provides programming access to GUI objects [2,4–8,24,124–126].

Accessibility technologies provide different aids to disabled computer users [127]. Specific aids include screen readers for the visually impaired, visual indicators or captions for users with hearing loss, and software to compensate for motion disabilities. Most computing platforms include accessibility technologies since electronic and information technology products and services are required to meet the *Electronic and Information Accessibility Standards* [127]. For example, *Microsoft Active Accessibility (MSAA)* technology is designed to improve the way accessibility aids work with applications running on Windows [128], and *Sun Microsystems Accessibility* technology assists disabled users who run software on top of *Java Virtual Machine (JVM)*. Accessibility technologies are incorporated into these and other computing platforms as well as libraries and applications in order to expose information about user interface objects.

Accessibility technologies provide a wealth of sophisticated services required to retrieve attributes of GUI objects, set and retrieve their values, and generate and intercept different events. Although MSAA is used for Windows, using a different accessibility technology will yield similar results. Even though there is no standard for accessibility *Application Programming Interface (API)* calls, different technologies offer similar API calls, suggesting a slow convergence towards a common programming standard for accessibility technologies.

The main idea of most implementations of accessibility technologies is that GUI objects expose a well-known interface that exports methods for accessing and manipulating the properties and the behavior of these objects [5–8, 24, 124–126, 129, 130]. For example, a Windows GUI object should implement the `IAccessible` interface in order to be accessed and controlled using the MSAA API calls. Programmers may write code to access and control GUI objects of GAPs as if these objects were standard programming objects.

Using accessibility technologies, programmers can also register callback functions for different events produced by GUI objects thereby obtaining timely information about states of the

GUI objects of the GAPs. For example, if a GUI object receives an incorrect input and the GAP shows an error message dialog informing the user about the mistake, then a previously registered callback can intercept this event signaling that the message dialog is being created, dismiss it, and send an "illegal input" message to the Designer that controls the GAP [2–4].

The idea as stated by Prof. Grechanik in his awarded NSF proposal in the working of a DbD strategy i.e. SEAPHISH platform is threefold: (1) A technique called defacing is employed where phishing is used to generate a fake app that resembles the original app that needs to be defended; (2) the original app is slightly modified to create additional uncertainty for the attacker, and (3) the installed app is monitored to determine if it provides useful services that are the reason for this app to be installed and how it uses resources.

Our assumptions are the following. First, the user downloads and installs an app that provides some useful services and it is the reason for using this app. For example, the user may be legally blind and s/he selects an assistive application that interprets and reads information from some other banking app to the user. If the downloaded app does not provide the service once it is installed, then it is removed and reported to the app store. Second, installed apps can freely use their resources to provide services like many security approaches that create sandbox environments around the installed apps. Consider our example with the assistive app that may send financial data from the smartphone to some external server to process this data and send back a WAV file to play to the user the transcribed financial data. This essentially means that you cannot prevent an app from using any external services that it may need as part of its functionality of providing assistance to the user. Thus, you cannot simply prevent an attack based on seeing that an app is performing some sensitive operation like reading/writing data from external entities. Third assumption is that the goal of the malicious app is to access external financial or safekeeping institutions to steal information or financial resources. That is, cases are not covered where a malicious application captures sensitive and personal data, like naked pictures and posts them on the Internet or collects some sensor information about the user's surroundings, e.g., recording and transmitting user's conversation. Next, defensive mechanisms "know" about the installed app and monitor its usage of resources. Finally, the installed app cannot analyze other applications and their environments including security certificates. The modified OS services would be used to enforce the latter assumption.

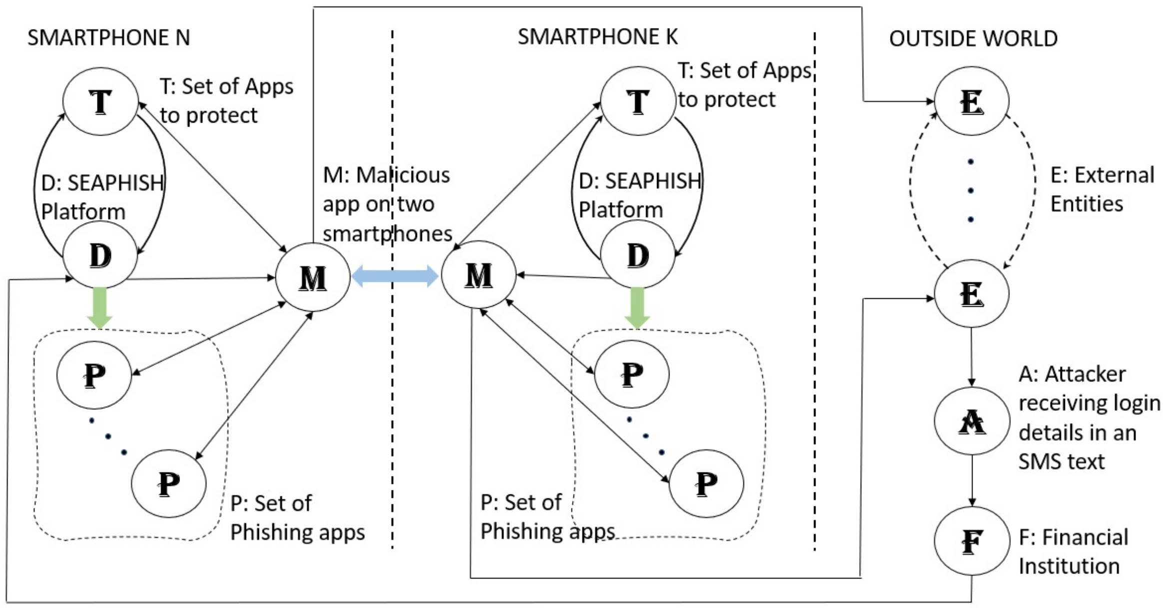

Figure 6: Working of SEAPHISH

Consider how SEAPHISH works using the diagram that is shown in Figure 6. The malicious app, M, is installed on multiple smartphones where for brevity we show only two, N and K

separated by the vertical dashed lines. The app is installed by the DbD SEAPHISH platform, D, whose goal is to evaluate the installed app and disable it if the app behaves suspiciously. A part of suspicious behavior is to overuse resources without providing services which are the reason for installing this app in the first place. Here, overusing resources could mean that the malicious app is utilizing memory and processing power for the purpose of performing an attack rather than providing any useful service. Of course, an attempt to log into the the financial institution, F with certain credentials known only to D and F is a sign of the malicious intent. If it happens, then the app M and all of its instances are disabled by D and reported to the app store.

The arrow between D and F depicts the collaboration between the DbD platform and the financial institution, F. It starts by F issuing a certain login and password that may look reasonable to an algorithm that attempts to analyze if it is machine or human-generated. For that, a collected database of passwords is used and the generated login/password pair does not match to any of the pairs that belong to F's users, and at the same time, this generated pair is statistically within the various ranges of login/password pairs from the database of passwords (e.g., the length of the password and the diagrams and trigrams are within the acceptable ranges). Besides D and F no entity knows about this generated login/password pair; any attempt to use it to log into a user's account with this pair will result in a lockdown of the account and disabling all of the instances of M installed on all smartphones.

Moreover, M does not have to connect to F directly to perform malicious operations – it can operate through multiple connected various proxies or external entities, E. Consider a situation where the stolen login and password are converted into text and M dials an external number and uses a voice narrator to leave a message on an answering machine that contains the login and the password. Then a different E will retrieve this message, convert it into text and will send this text as an SMS to a phone of an attacker who will actually attempt to log into the F's website. In SEAPHISH, no matter where the information comes from the end goal is the attack on the website.

The input to D is the list of all apps installed on the smartphone that must be protected, e.g., banking or general financial apps, house security control apps, or health record apps. The first step that D performs is to create a set of phishing apps for each of the protected app, T. That is, it uses the accessibility layer to mimic how users interact with the app, T, by performing various actions on its GUI object to collect the layouts of screens, data, and to construct the state machine graph where the nodes are the screens with GUI objects and their values and the edges designate transitions between these screens. The captured state machine is used to generate a phishing app, P, which is a key of the proposed DbD strategy. Of course, the resulting phishing app has the limitation of how deep D expores the real app, since creating a fake replica of the real app is an undecidable problem. This is a part of this DbD game where M is using its analysis to determine whether an app it interacts with is T or P. If M makes the type 1 error by mistaking T for P i.e. a false negative, then it will keep wasting its analysis resources until D exhausts work credit it gives to M and marks it as malicious. Conversely, if M makes the type 2 error by mistaking P for T i.e. a false positive, then it will capture the fake login that D uses to log into F and attempts to to use it whereby it will be detected by F and neutralized.

To confuse the malicious app that T is the actual victim app and not one of the Ps, the platform D modifies T by injecting new GUI objects into its GUI layouts and by changing the geometry and various properties of the existing GUIs. These modifications do not change the functionality of the app T, however, they increase the complexity of the analysis that M must perform. Since creating P by simply cloning T does not affect M that can perform an attack on the clone on T, generating P in conjunction with modifying T requires the cooperation of the underlying OS to ensure that its security mechanisms do not reject the generated Ps and the modified T. At the same time, the OS must have information about the protected apps, T and the new app, M, so that it can detect and prevent M from accessing the code and certificates for T and Ps. This is done by modifying the OS and ensuring that no permission given by the user can enable M to access these specific services unless of the following two events happen: the user goes into the OS with binary rewriting tools in the developers mode to enable M to access these services or M passes the trial period and it is given the permission to access these services as a trusted app. As a result, the only way for M to determine which app is T and which ones are P is to keep providing useful services for the smartphone user while exploring the GUI state machine to determine discrepancies with the version of T that the attackers who created M explored.

Thus, for the SEAPHISH platform to modify an app's GUI layouts in order to deceive the malicious app, the platform should simulate conditions as seen in the real world apps. The platform should know the ideal number of screens or activities seen in an app that should be simulated. Thus, for an app consisting of structured GUI objects, when entering data into a simulator for SEAPHISH, how would the platform know whether an app can have 10, 50 or 500 screens? For a screen, again what should be the number of GUI objects that could be present? Such questions can be answered with a statistical record maintained for GUI objects and their properties seen in Android apps. Without such statistical information, developing an effective simulation for SEAPHISH can be a very difficult problem.

CHAPTER 3

OUR CONTRIBUTION

As the best way to enhance SEAPHISH simulation is to obtain statistics on the properties of GUI layouts of apps at large, we investigate possible solutions in generating the GUI model of an app. Specifically, our contributions are as below:

1. We investigate various state-of-the-art tools that use different strategies in automatically traversing through the different states of an app. We use the available documentation for each of the shortlisted open source tools and test it on a couple of sample apk files to check the following:
    - Does the tool work successfully in generating an output?
    - Does the generated output help in obtaining a model of the app's state machine? If not, what enhancements need to be done to the existing tool?
2. Once we have the tool that can generate the required models, we test it on a small collection of 200+ apk files.
3. We next design a framework that can automatically obtain the GUI details of a large collection of 3 million Android apps.
4. We statistically analyze the resulting models and obtain details on various GUI objects and screens.

3.1 Challenges

The main challenge in this work is exploring the existing tools and finding one which meets our criteria with little effort and cost. There are eighteen open source tools that we explore and discuss in Chapter 4. Although the literature provides relevant details of each tool, the project documentation on steps and requirements to run each tool is limited in many cases. Getting each tool to work and analyzing the outputs has certain time and effort associated with it.

In cases where the metadata for an apk file is available in XML files, we hope to save on the runtime expense by simply analyzing the XML metadata to construct the GUI model rather than run the app on an emulator or device. However, we would need to rely on the tool's effectiveness in best handling the case where only the app's default layout is declared which then gets modified unpredictably at runtime.

When using a tool which relies on dynamic analysis i.e. finding paths by running the app on an emulator or device, there could be the problem of combinatorial explosion which arises due to the possibility of multiple different paths across different GUI objects/screens. This may cause the tool to run for a significant amount of time in performing the analysis which may not even be near to completion.

CHAPTER 4

RELATED WORK

With the tremendous growth in usage of Android apps and Android being open source, there is a lot of focus towards testing Android apps efficiently so that they meet their functional requirements in a qualitative manner. Manual testing of an app can be a laborious task and it may not uncover all the defects in a timely and cost-effective manner. Instead, there is growing research [131, 132] towards developing tools that can perform automatic testing of an app. These tools follow different strategies and aim towards detecting different kinds of bugs in apps like detecting bugs that do not meet functional requirements [133] or those that result in memory leak patterns [134] or detecting energy inefficiencies [135] or performance related bugs [136]. One of the most expensive tasks in the area of software testing is generating test input [131].We explore different tools in this chapter that aid with the automatic testing of Android apps. We investigate whether these state-of-the-art tools can help us in determining the finite state machine of an app in an efficient way in terms of cost and effort.

4.1 Dynamic Analysis

This section covers various tools that perform dynamic analysis of Android apps that generate test inputs by mimicking user interactions or events like clicking a button, entering values in text fields etc. This set of tools are classified with regards to their exploration strategy and we check whether these could or could not be used for the purpose of our thesis.

1. **Random Exploration Strategy or Fuzz Testing:**

   Fuzz testing is an approach used to automate the process of testing software by a program called a fuzzer, which generates a large amount of input data for the target program. This technique is used to detect unknown vulnerabilities [137]. The drawback with fuzz testing is that since inputs are tested randomly, code coverage is generally low. Consider, for example, a simple conditional statement: "if (a == 1357) then {...}". Here, 'a' is a randomly chosen 32-bit value. Now the 'then' branch is exercised only when the value for a is exactly 1357. Thus, it has only one in 2^{32} chances of being exercised [138, 139]

   The set of tools that fall in the Random Exploration/Fuzz testing category randomly generate GUI events. This strategy may be apt in stress testing as random exploration can continually trigger events till some manually specified timeout is reached. However, as the exploration is random, the tools may generate the same event multiple times and there may not be a clear indication that all possible events across the different GUI screens have been covered within the timeout duration.

   (a) **Monkey**

   The UI/Application Exerciser Monkey commonly called 'monkey' is a command-line tool provided by the Android integrated testing framework. This tool can be executed on an emulator instance or on a device by generating and sending a pseudo-random stream of user events like keystrokes, clicks to the system [140].

   As monkey is included in the Android developer's toolkit, installation effort is minimum. Based on the selected verbosity level, reports on the progress and events generated by monkey can be obtained. However, the tool can only implement a random strategy while exploring the app. While running the tool for a particular Android app, the user needs to specify the number of events to be generated. Once this limit is reached by the tool, the exploration stops. Thus, the random strategy is not guaranteed to capture all details across all GUI screens for an app and hence we did not consider this tool for our thesis.

(b) **MonkeyRunner**

The monkeyrunner [141] tool can control an emulator instance or device outside of Android code by programs written through an API. It assists functional testing by automatically running an entire script that tests the application. Thus, a program can be written in Python to install the app that needs to be tested, run the app and provide inputs in terms of keystrokes or touch events to the app. The tool allows one to capture and save a screenshot of the user interface. The monkeyrunner API can also be extended with user defined classes.

However, as the input should be provided via a program, GUI details of the app must be known in advance to running monkeyrunner on the app. Hence, this tool was not considered for automatically analyzing the GUI state model of apps.

(c) **Dynodroid**

The working of dynodroid [142] tool relies on an observe-select-execute cycle wherein for the app's current state, the tool first detects the relevant events. For this, the tool detects for the current screen, the GUI layout of objects along with the kind of input expected by each GUI object. In the select stage, the tool selects one of the observed relevant events using one of three different strategies – Frequency, UniformRandom and BiasedRandom. The frequency strategy picks up the least frequently selected event from the set of possible relevant events. The UniformRandom strategy selects an event uniformly at random. The BiasedRandom strategy makes use of a context-sensitive model such that events which are applicable to more contexts would be selected more frequently. Finally, in the execute stage, the tool executes the selected event which may result in a new state after which it again repeats this cycle.

In comparison to Monkey, Dynodroid provides additional features like being able to compute relevant events, generate system as well as GUI events. However, while trying to download and work on the tool from the publicly available site [143, 144], we found that the Google drive links were outdated. Thus, we could not successfully install the tool and test it on sample apps.

2. **Systematic Exploration Strategy**

This set of tools systematically tests an app by executing it symbolically. Symbolic [145] execution divides the complete set of inputs into different classes such that each class corresponds to a unique program behavior. To understand symbolic execution, consider the same example used earlier for fuzz testing i.e. consider a simple conditional statement: "if (a == 1357) then {. . . }". Here, symbolic execution collects the constraints on inputs from conditional statements that it finds in the program. Thus, symbolic execution of the example conditional statement for input $a = 0$ produces the constraint $a \neq 1357$. Once this constraint is negated and solved, it will yield a = 1357, a new input that leads to new program behavior that follows the 'then' branch of the conditional statement [138, 139].

Thus, instead of using random techniques, it uses a systematic strategy to generate specific, non-redundant inputs. A drawback of this type of exploration is that it is considered less scalable as the complete set of input classes required to cover all possible program behaviors is essentially infinite leading to the path explosion problem [142].

(a) **ACTEve**

ACTEve (Automated Concolic Testing of Event-driven programs) [146], is based on concolic testing (also referred to as dynamic symbolic execution) and automatically produces sequences of events in a systematic manner. It uses the concolic method of execution to symbolically monitor events right from where they are produced up to the point where they are handled and processed. Concolic testing [147] uses a combination of symbolic and concrete execution to produce test inputs that can explore all possible execution paths. Building on our previous example used for fuzz testing and symbolic execution, consider an additional conditional statement as below. Here b is an integer variable.

```
if (a == 1357) then {
  if (a < b) then {
    ...}
    }
```

We start with an arbitrary choice for the two variables ‘a’ and ‘b’. Now symbolic execution would solve the constraint and yield a = 1357 to reach the ‘then’ branch of the first conditional statement. Now, if b was chosen to be equal to 1, then the second conditional statement would fail as 1357 (a) is not less than 1 (b). Thus, the path conditions are a = 1357 and a >= b. The second condition is negated giving a < b. Thus, in this case, concolic execution approach would involve looking for values of a and b such that a = 1357 and a < b; for example, a = 1357, b = 1400. Hence, the input value of b = 1400 will satisfy the second conditional statement and the ‘then’ branch of the second conditional statement can be explored.

ACTEve instruments both the Android framework and the input app under test and tries to assuage the path explosion problem by handling program executions in a way that helps with pruning redundant event sequences.

Although the tool is publicly available [148], we did not find relevant information on the steps required for setting up and testing the tool on sample apps. Due to unavailability of proper documentation, we did not explore this tool further.

(b) **Sapienz**

Sapienz [149] is a multi-objective, search-based automated software testing tool for Android apps. Sapienz combines the random, systematic and search-based exploration strategies. Sapienz makes use of motif patterns, a set of patterns of low level events which are good indicators of achieving high coverage based on the current screen’s GUI information. A primary focus of Sapienz is to analyze and optimize length of test sequences, while at the same time, maximize program coverage and fault detection.

However, the GitHub page of the online Sapienz prototype [150] mentions that the prototype is out-of-date and no longer supported.

(c) **EHBDroid**

EHBDroid (Event Handler Based) [151] is an automated, open source testing tool for Android apps based on the Soot framework [152].

"Soot is used to analyze, instrument, optimize and visualize Java and Android applications" [153]. Soot builds a control flow graph that represents the intra procedural data flow analysis. In this graph, the nodes depict the program statements and an edge between two nodes A and B indicates the flow of control from the statement depicted by node A to the statement depicted by node B [153–155].

We will later see another tool GATOR that is described in Section 4.2 Static Analysis that is also based on the Soot framework.

Now, EHBDroid does not produce events from the GUI but instead directly triggers callbacks of event handlers, unlike other traditional GUI testing methods. This helps EHBDroid simulate a larger set of events that are otherwise not easy to produce by conventional GUI-testing methods.

EHBDroid consists of two basic components: 1) an **Instrumentor** that instruments the input app. Here, a relevant sequence of callback invocations in each activity is collected by analyzing various unique event registration patterns found in XML resource files along with the app's program code 2) a testing **Explorer** that tests the target app. As a list of analyzed activities is tracked, the Explorer detects if the present activity's exploration is completed. On encountering any unexplored activity, the Explorer will automatically analyze the current activity else it returns back to explore the currently top of the stack activity [156]. This process continues until exploration of all activities in the activity stack is complete.

EHBDroid is publicly available [157]. However, although we were able to instrument a sample app after setting up and running the tool; further steps of signing and installing the app led to multiple exceptions and we were unable to resolve the errors.

3. **Model-based Exploration Strategy**

This set of tools focus on automated GUI testing of Android apps by building a model that abstracts the app's behavior. Using the model, the testing tool then derives tests in order to validate an app's functionality. The built models often represent finite state machines where activities denote the states and events denote the transitions. This set of tools dynamically build a model till all possible events that can be generated by all explored states of an app have been analyzed.

However, producing a complete set of tests based on an abstract model of the app can get overwhelming. Depending on the number of screens, GUI objects and possible transitions between the objects, the problem of path explosion can make it quite difficult to obtain and execute all possible tests.

(a) **SwiftHand**

SwiftHand [158] makes use of machine learning to learn and build a model of an app while testing the app and uses the built model to generate test input sequences which automatically analyze unexplored states of the app. Based on the execution results, the tool then refines the model. One of the important features of SwiftHand is to optimize the exploration strategy by minimizing the need to restart the app under test. This feature is considered important as traditional exploration techniques often restart an app by removing and installing it again in order to explore all states reachable from the starting state. However, restarting an app is considered a time consuming task rather than just executing a sequence of inputs on an app. Another focus of this tool is to realize code coverage rapidly by learning and analyzing the built abstract GUI model of the app. This is achieved by relying on the principle that instead of following a computationally expensive approach towards building a precise model of the app, an approximate model of the app can be assumed to suffice in guiding the generation of test inputs.

We followed the steps provided on the publicly available site [159] for building the tool, however, we were not successful in getting it to work on a couple of sample APK files. Hence, we did not explore the tool further.

(b) **PUMA**

PUMA [160] is Programmable UI-Automation Framework for Dynamic App Analysis. It includes a basic random exploration strategy like Monkey and exposes an event driven programming abstraction. Thus, the basic framework can be enhanced to support any dynamic analysis on Android apps. The exploration strategy in PUMA can be modified by implementing compact event handlers that separate the logic between analysis and exploration. The tool is publicly available [161] but this has been tested only on Ubuntu machine (12.04), Android (4.3). As the tool is compatible only with a specific release of the Android framework, we did not explore this tool further.

(c) **Stoat**

Stoat (STOchastic model App Tester) [162] is an automated testing tool that tests whether an app meets its desired functionality by triggering different user and system events. Stoat works in two stages: First, the GUI layouts of the target app are expressed as a stochastic model which is a finite state machine with edges linked with probabilities for test generation. Specifically, Stoat builds the stochastic model by exploring different behaviors using a dynamic analysis strategy that is extended by a weighted GUI exploration strategy and static analysis. Second, in an iterative manner, Stoat mutates the built stochastic model producing tests from the model mutants. By perturbing the probabilities of edges, Stoat can produce unique tests to detect deep bugs that would otherwise not usually be detected.

The tool is publicly available [163] where the implementation has been tested with Android 4.4, running on Ubuntu 14.04. However, this version only supports testing ant projects. Hence, we did not explore this tool further.

(d) **A^3E**

The A^3E (Automatic Android App Explorer) [164] toolset provides automated GUI testing of Android apps by modeling an app in the context of a Static Activity Transition Graph that depicts transitional relations between activities [165]. For example, a directional edge from activity 1 to activity 2 indicates a possible transition from activity 1 to activity 2.

Using the Static Activity Transition Graph, A^3E [166] can implement two kinds of exploration strategies. **1) Targeted Exploration** focuses on a fast, direct strategy of traversing between activities, including those activities that are less likely to be normally explored. It can detect activities that are triggered from other apps or services without requiring user interaction. **2) Depth-First Exploration** uses a systematic way of mimicking user behavior by navigating from one screen to the next. The traversed screens are pushed into a stack so that the screen at the top of the stack is popped if a return event to a previous screen is triggered.

Only the depth-first implementation of the tool is publicly available and not the targeted version. As per the GitHub page [167] of the depth-first version, it is tested under Ruby 2.0 and Android 4.4.2. Since this version of A^3E seemed to require major efforts in setup and as it is compatible only with specific versions of Android, we did not explore this tool further.

(e) **DroidBot**

DroidBot [168] is a light-weight, UI guided test input generator that produces inputs using a state transition model built during run time along with combining user defined testing algorithms and methods. The generated model is essentially a directed graph where each node denotes a state of the app that includes the GUI details and each edge between two nodes denotes the input event that triggers the transition along with its methods and log details. The default exploration strategy used for producing the test inputs is depth-first. Droidbot also provides a set of high level APIs that allow customized scripts and algorithms that enhance the event generation modules to be included thereby making Droidbot an extremely extensible tool.

We were able to run the publicly available tool [169] on sample APK files. First, we installed an APK file in an emulator using the command **'adb install sampleApk.apk'** where sampleApk.apk is an Android apk file. Next, we ran the droidbot tool command **'droidbot -a sampleApk.apk -o output_dir'** where -a is the argument for file path to the apk file and -o is the argument for the directory of output. The command allowed the tool to start the app on the emulator from the Main activity. However, we were not able to get the tool to automatically provide inputs to the text fields on the main screen, without which the app could not proceed to the next screen. Now for our thesis work of analyzing up to 3 million apk files, we needed a tool that could automatically provide inputs when needed, so that the different activities in the app could be automatically explored. Thus, if we were to use droidbot for our thesis work, we would need to investigate and maybe write customized scripts such that when the tool detects an input field on the current screen, it would need to generate the inputs.

(f) **AndroidRipper**

Android GUI Ripper [170, 171] automatically explores an Android app by exercising its GUI in both systematic exploration strategies (Depth First and Breadth First) and Random strategies.

The AndroidRipper tool uses a GUI crawling based approach to dynamically construct a model of the input app from an initial state. On exploring a new state, a list of events that can be triggered from the current state is tracked and the events are systematically invoked. When no new states can be detected during the exploration, the tool restarts the process from the initial state. Thus, AndroidRipper is mainly based on ripping feature which uses the crawler to simulate real user interactions to automatically and systematically traverse the app's screens, build a GUI model, produce and run test cases based on the model as new events are found [172–174]. Figure 7 depicts the iterative algorithm used by the GUI crawler in building the model [10].

We were able to configure and set up the environment as needed to run the latest version [175] of AndroidRipper on input APK files. The tool automatically traverses through the screens in the app and generates model of the GUI in xml format. As no manual intervention was needed to run this tool on an app, we selected this tool

0) Describe the starting interface (associated with the first interface shown by the application at its launch) in terms of its activity instance, widgets, properties and event handlers, and store this description into the Interface list;

1) Detect all the interface fireable events having an explicitly defined Event Handler and, for each event, define a possible way of firing it by choosing the random values that will be set into the widget Editable Properties and to the Event Parameter Values (if they are present). Save this information into an Event description and store this description into the Event List[1].

repeat
2) Choose one fireable event *E* from the Event List, set the needed preconditions and fire it, according to its description.

3) Catch the current interface and add a node representing that interface to the GUI tree; then add an edge between the nodes associated with the consecutively visited interfaces.

4) Describe the current interface in terms of all its properties, store the interface description in the Interface List, and check whether the current interface is 'equivalent' to any previously visited one, or it is a 'new' one. If it is equivalent to any interface or it does not include fireable events, the corresponding GUI node will be a leaf of the tree, otherwise the new interface fireable events will be detected and a description of each event will be defined and added to the Event List. In both cases, the E Event that caused that interface to be reached will be removed from the Event List.

Until the fireable Event list is empty

Figure 7: The Crawling Algorithm from [10]

for performing dynamic analysis of the app to help generate the GUI state machine.

Chapter 5 provides more details on the output produced by AndroidRipper.

4.2 Static Analysis

In the previous section 4.1, we explored various tools that perform dynamic analysis of an Android app. However, such an analysis would require running the app on an emulator or Android device which can be a substantially expensive operation and it may be the case that even after spending a significant amount of time, the analysis may not obtain high coverage [131]. As Android development guidelines suggest that separating the design of the GUI layout in XML files from the code that implements its functionality is best practice [176], we explore another set of tools that perform static analysis of Android apps such that without running the app, these tools are able to reconstruct the GUI state machine by simply analyzing the XML metadata, thus saving the runtime expense.

1. **Flowdroid**

   Apps may accidentally leak sensitive data or malicious apps may exploit permissions granted by the user to deliberately copy sensitive data. Flowdroid [177] focuses on spotting such data leaks by performing a static taint analysis of Android apps.

   Taint analysis involves providing possible malicious data flows to malware detection tools or human specialists who can further inspect whether a leak in fact corresponds to a policy violation. This analysis method detects possible leaks by monitoring sensitive "tainted" flow of data initiated from a source (for example, an API method that returns a message containing financial information) and tracks it till it reaches a sink (for example, a method that writes this message to an external server).

"A precise model of Android's lifecycle allows the analysis to properly handle callbacks invoked by the Android framework, while context, flow, field and object-sensitivity allows the analysis to reduce the number of false alarms." [177] However, Flowdroid handles lifecycle/callback events and customized GUI objects only for a single activity. Hence, we did not further consider Flowdroid as our thesis involves apps which may be composed of multiple activities and efficiently traversing through multiple activities in generating the GUI model was an important requirement of our thesis.

2. **GATOR**

   Gator (proGram Analysis Toolkit fOr andRoid) was first publicly available in Feb 2014 and since then there have been total 15 versions [178] released with changes related to including new features, performance enhancements, optimizations, bug fixes as well as support to run as a Docker container. Gator takes as input an APK file and runs analysis on top of the Soot program analysis framework. We saw a short description of the Soot framework earlier while describing the tool EHBDroid in Section 4.1 Dynamic Analysis.

   In Gator, the first component focused on modeling the static flow of object references in a program [176]. This considered the possible data and control flow based on the GUI objects and the various interactions between them by the corresponding event handlers. Thus views, activities and listeners are abstracted and a constraint graph depicting the structural relationships consisting of views corresponding to activities and listeners and hierarchical structure of parent-child view relationships is developed.

The next component of Gator built on previous work, focuses on a program representation that helps with carrying out graph reachability by navigating context relevant interprocedural control flow paths detecting program statements that could invoke callbacks along with paths that do not include such statements. Here, callbacks are of two types: Lifecycle callbacks and GUI event handler callbacks.

Lifecycle callbacks: The Activity instances in the app shift through various phases in their lifecycle depending on how the user navigates through an app [179]. The Activity class offers six main callbacks: onCreate(), onStart(), onResume(), onPause(), onStop(), and onDestroy() which lets an activity understand when there is some change in the state such as when the system creates, stops or resumes the activity or destroys a process where an activity exists.

GUI event handler callbacks: These are associated with various user interactions. For example, when an object like button is clicked, the onclick() method is called on the object [180]. Such event handlers function different actions that lead to various transitions in the app logic like terminating an activity or returning to a preceding activity.

Subsequent work led to building a window transition graph (WTG) that denotes potential sequences of GUI windows with their corresponding callbacks and events [181]. This work focuses on modeling a window stack to monitor the active windows at present, updates to the stack with effects of callbacks.

Further work led towards detecting energy defects [182], exposing resource leak defects [183] and a responsiveness profiling strategy that expresses response times as a function of the size of a possibly expensive resource [184].

We ran version 3.8 (released in September 2019) [185] of Gator on a sample APK file as a docker container which did not require any manual setup. We just had to build the docker image and run the container which provided us with information on the WTG namely, the stack of active windows and changes to the stack. However, we were not able to extract the GUI model of the app in a format like XML or JSON that we desired. Hence, we did not consider this tool.

3. **Backstage**

Backstage [186,187] is an open source automated static analysis framework designed with the aim of finding irregularities between the expected behavior of a GUI object on a screen versus the actual behavior implemented by the GUI object. For example, an app 'BMI Calculator' that calculates Body Mass Index (BMI) has a screen with two text fields which accept height and weight parameters as input and has a button 'Calculate BMI'. Now, based on the GUI objects on this screen, the user would expect that after entering the height and weight details, clicking the 'Calculate BMI' button should only return the calculated BMI value. However, if the button has additional functionality that also sends the user's location to an external service, then such a behavior needs to be detected as irregular and reported. This may be deliberately done by those who want to attack the security and privacy of users or also maybe unintentionally done by developers who provide additional functionality that is inconsistent with the expected behavior of the GUI objects that users see on screen. Backstage aims at finding such anomalies in apps. The functionality of the tool as implemented [188] consists of 2 stages: 1) GUI Analysis Phase 2) Detecting outliers phase.

Stage 1) GUI Analysis Phase: In this phase, Backstage first analyzes behavior of the app by investigating the GUI objects specified in the app's xml layout files together with the corresponding code. Next, the tool constructs a control flow model by using the event handlers corresponding to the lifecycle and interactions of the GUI objects. Lastly, the code associated with the callback functions is analyzed to determine the content of GUI objects. Backstage can obtain the text of GUI objects in multiple ways like through the Android:text attribute specified in XML files, through "@string/" prefix that can be used to reference the app's resources or simply the direct string that will be displayed, through styles.xml file where the label of a GUI object may change based on the style, through code (example method View: setText(text) defines the text for GUI objects) or through the text defined in icons or alternative text, which is specified in the android:contentDescription attribute of the GUI object.

Stage 2) Detecting outliers phase: Backstage uses the callbacks from stage 1 as potential entry points and builds a call graph via Rapid Type Analysis algorithm (RTA) that restricts the over-approximation in the analysis through determining potentially instantiated classes [186, 189]. Using methods reachable from the callbacks, Backstage tries to determine which sensitive API calls [190] can be invoked. Here sensitive API calls are those calls that can perform operations like accessing user's location or which can carry out operations like sending messages to external services etc.

We ran backstage on sample apk files. The output of stage 1 of the tool provided the GUI model of the app in XML format depicting the hierarchy of the GUI objects and information on the different XML layout files could be extracted. We did not run the second stage of the tool as we did not need to detect any outliers corresponding to the GUI objects. On analyzing the output files obtained from stage 1, we could see that this tool could provide us the required information in the most cost-effective way. Chapter 5 provides more details on the output produced by Backstage.

4.3 Hybrid Analysis

This section consists of a set of tools that perform both static and dynamic analysis of an app's GUI and we investigate if any of these tools could be used in our thesis.

1. **Orbit**

   Orbit [191] uses a grey-box approach that combines static and dynamic analysis for Android apps. In the first stage, on performing a static analysis of the app's code, the tool extracts a group of actions that can be invoked by each GUI object. In the second stage, dynamic analysis based on a crawler reverse engineers the built model by systematically invoking the actions on the running app. However, since this tool is not publicly available, we did not explore it further.

2. **MonkeyLab**

   MonkeyLab follows a record-mine-generate-validate approach. "This framework relies on recording app usages that yield execution (event) traces, mining those event traces and generating execution scenarios using statistical language modeling, static and dynamic analyses, and validating the resulting scenarios using an interactive execution of the app on a real device" [192]. Thus, the analysis provided by this tool depends on first recording usages of an app. As our thesis involves analyzing up to 3 million apps without having any usage details of the apps, we did not consider this tool.

3. **Amoga**

   Amoga (Automated MOdel Generator for Android apps) [193] uses a hybrid strategy of first using a static analyzer that constructs the WTG of an app using Gator. In this stage, the apk file is decompiled to obtain the bytecode and then a control flow analysis is performed. In the second stage, a crawler built on top of the Robotium framework [194] invokes the events detected from the static analysis stage, in a sequence at runtime to transition between different states of the app. As we could not find this tool online, we did not explore this further.

CHAPTER 5

SOLUTION

Of all the tools that we reviewed in chapter 4, we found two tools - Backstage and AndroidRipper, that were useful in helping us extract a model of the app's GUI in XML format. Backstage uses static analysis while AndroidRipper uses dynamic analysis. Using these two tools, we design and propose a hybrid framework in section 5.1 and then we describe the outputs obtained on running the tools, Backstage and AndroidRipper in sections 5.2 and 5.3 respectively.

5.1 Proposed Framework

Figure 8 depicts the proposed framework for analyzing GUI layouts of input APK files using the tools Backstage and Android GUI Ripper.

1. As input, we have a large dataset of about 3 million APK files, each having filename as a 16 bytes Globally Unique IDentifier (GUID). We form a list of the APK file names from this collection and parallelize processing of these files further in the pipeline.

2. We build a resource manager component that computes the available memory and processing power in order to tune the level of parallelism across many Virtual Machines (VMs). Various metrics that are collected are average CPU utilization, memory utilization or disk utilization. The resource manager balances the workload among the virtual machines which can be scaled in or scaled out depending on the computed metrics. Here,

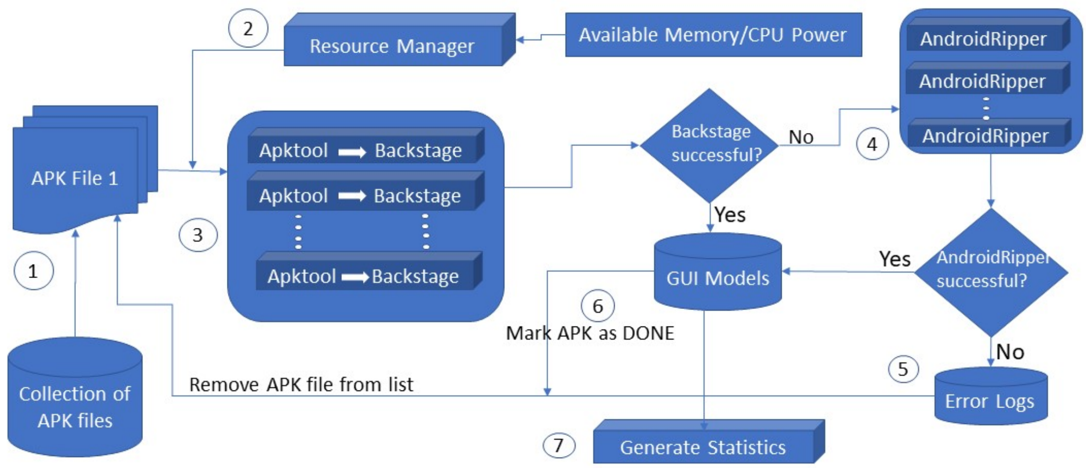

Figure 8: Proposed Framework using both the tools - Backstage and AndroidRipper

workload refers to processing of each input apk file in the pipeline. The resource manager component is required to achieve a balanced resource utilization and to optimize overall performance by proper use of available resources.

3. For each input apk file, we need to run the decode option of apktool command to extract the XML resource and layout files. Using these files as input, we next run Backstage to perform static analysis and generate the GUI model of the app.

4. Apk files for which Backstage fails to extract a model are flagged for further processing. Again, based on computed memory and processing power metrics by the resource manager,

the Android GUI Ripper can be made to run on these flagged APK files in parallel. This phase would include running the app on the emulator to dynamically extract model of the app.

5. We log errors generated in step 4 in log files and remove the corresponding apk files from the input collection.

6. Apk files for which either Backstage or Android GUI Ripper provides GUI models in xml files are stored and marked as done and removed from the input collection to avoid any further processing.

7. From the xml files extracted in stage 6, we run our statistics script to extract different statistical results like number of clickable objects, scrollable elements, input fields, screens/activities and layout files. We calculate various metrics like mean, median, mode, standard deviation, variance, proportion, range, skew, rank etc. for the GUI objects extracted from the apps.

   Thus, our framework focuses on developing a solution that supports fault tolerance, scalability and optimized performance.

5.2 Static Analysis - Backstage

We found Backstage to be the most relevant for our thesis as it uses static analysis to extract a model of the app's GUI in XML format, thereby eliminating the run time expense. We downloaded the source code from GitHub [188] and built the Maven [195] project to produce a jar file. We used a sample apk file **'com.zola.bmi.apk'** which is the BMI calculator Android

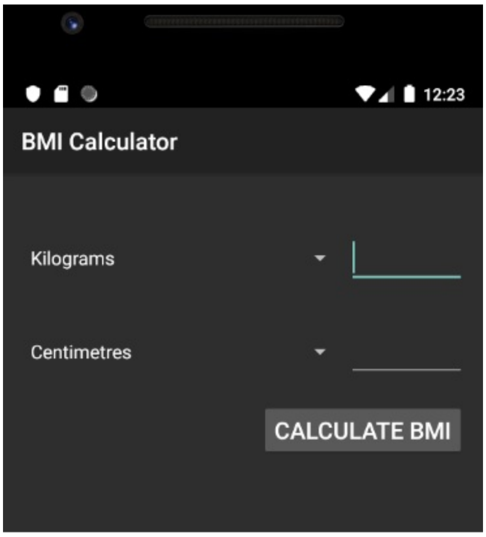

Figure 9: Main Screen of BMI Calculator App

app. This app takes input via two text fields which represent weight and height and on the click of a button 'CALCULATE BMI', it displays a message with the BMI value. Figure 9 shows the entry screen of the app.

We first run the apktool command with decode option on the file 'com.zola.bmi.apk' to extract the XML layout and resource files as shown in Figure 10. The res folder further

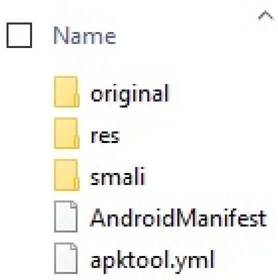

Figure 10: Files extracted on decoding input apk file using apktool command

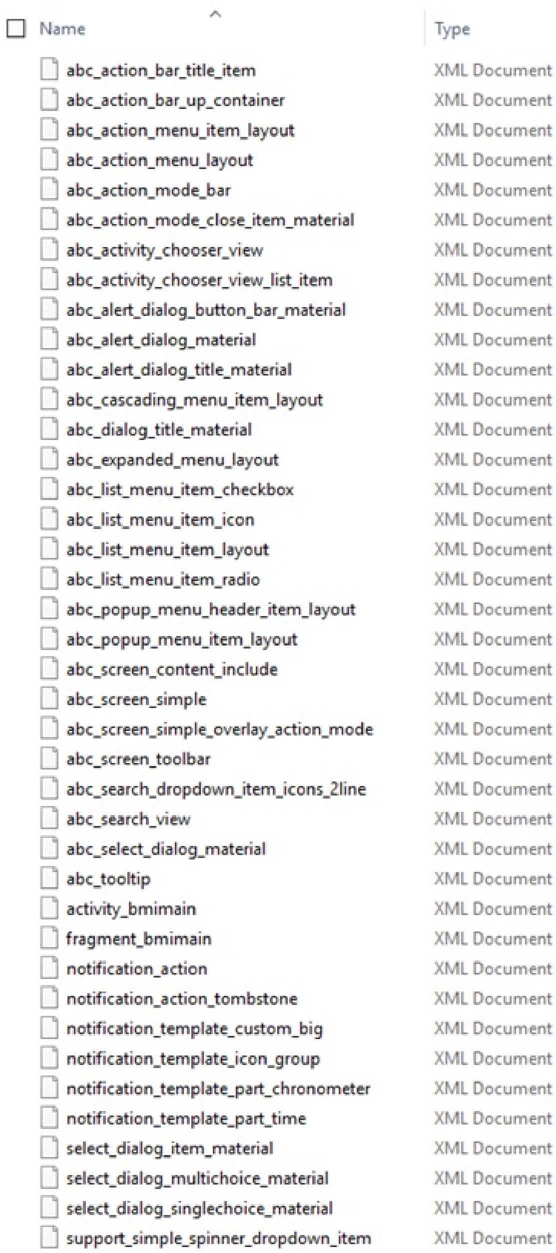

Figure 11: XML Layout files extracted from input apk file

contains different files and folders and in this, the layout folder contains multiple xml resource files as shown in Figure 11.

The backstage command is next run as below [196]:

"java -Xmx40g -Xss5m -cp target/Backstage-5.1-SNAPSHOT-jar-with-dependencies.jar st.cs.uni.saarland.de.testApps.TestApp -apk output/com.zola.bmi.apk **-androidJar libs/android.jar -apkToolOutput** output/com.zola.bmi **-rAnalysis -uiTimeoutValue** 30 **-uiTimeoutUnit** SECONDS **-rTimeoutValue** 30 **-rTimeoutUnit** SECONDS **-maxDepthMethodLevel** 15 **-numThreads** 24 **-rLimitByPackageName"** where

- **-apk** *.apk refers to location of .apk file
- **-apkToolOutput** * refers to the output folder containing app GUI models extracted by Backstage
- **-rAnalysis** indicates running UI analysis
- **-uiTimeoutValue, -uiTimeoutUnit, -rTimeoutValue, -rTimeoutUnit** indicates running UI analysis with timeout duration for entrypoint class
- **-maxDepthMethodLevel** indicates number of hierarchy levels that should be searched around the id element
- **-numThreads** indicates number of threads for running the analysis
- **-rLimitByPackageName** limits the analysis to classes based on the package name. For example, to analyze the Yelp app, classes belonging to only the com.yelp package would

Figure 12: Tree view of GUI model in appSerialized.txt file

be considered. This is done to eliminate the third party libraries that are often included in apk files.

On running the Backstage command, in the output folder, file **'appSerialized.txt'** which represents the GUI model of the app is generated. Figure 12 shows a tree view of the main XML tags present in the appSerialized.txt file. Details of one GUI element – 'Calculate BMI' button is shown in Figure 13.

The important tags that can be seen in file 'appSerialized.txt' are:

- <activities>tag as seen in Figure 14 lists out names and labels of the activity
- <kindOfUiElement>as seen in Figure 13 indicates the type of UI element like Button, TextView etc

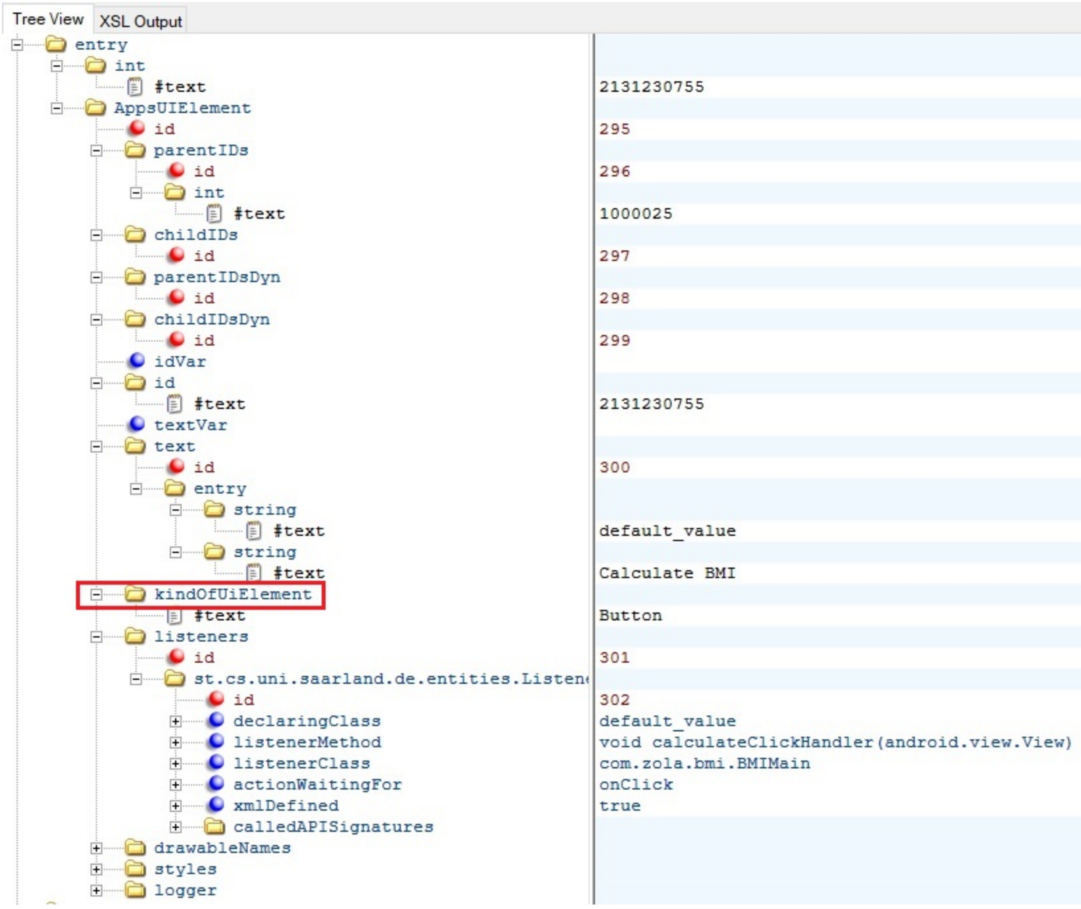

Figure 13: Part of the GUI model for element 'Calculate BMI' button

- <uiElementsOfApp>as seen in Figure 15 consists of parent-child relationship details for the ID of the GUI elements referred to by <int>along with any <styles>defined

'xmlLayoutFiles.txt' is another file extracted by Backstage tool which provides ids of GUI objects seen in each resource layout file for the app. Figure 16 shows a part of the xmlLayoutFiles.txt file.

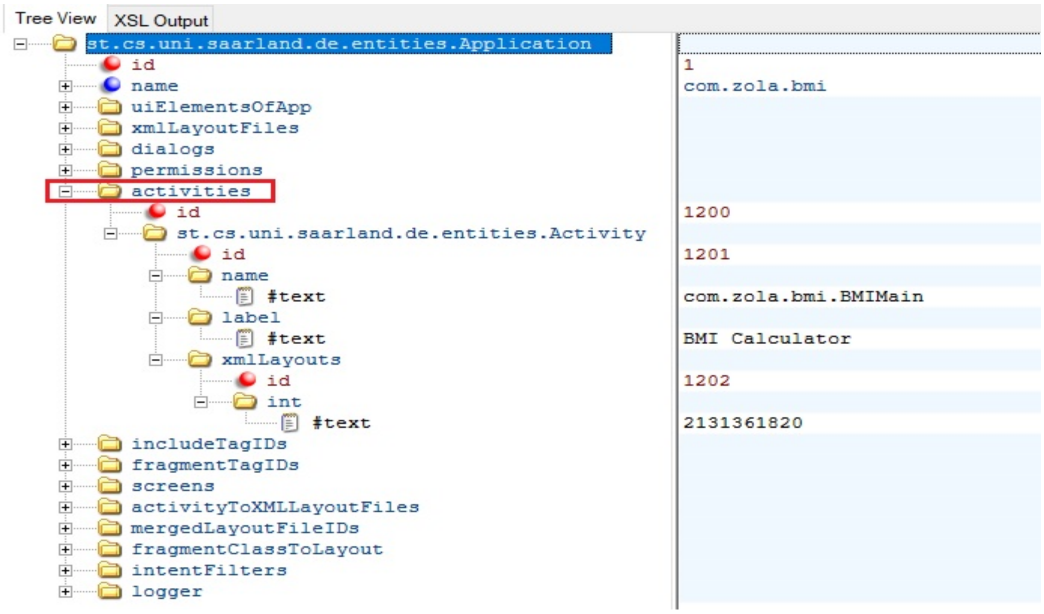

Figure 14: Details of activities tag in appSerialized.txt file

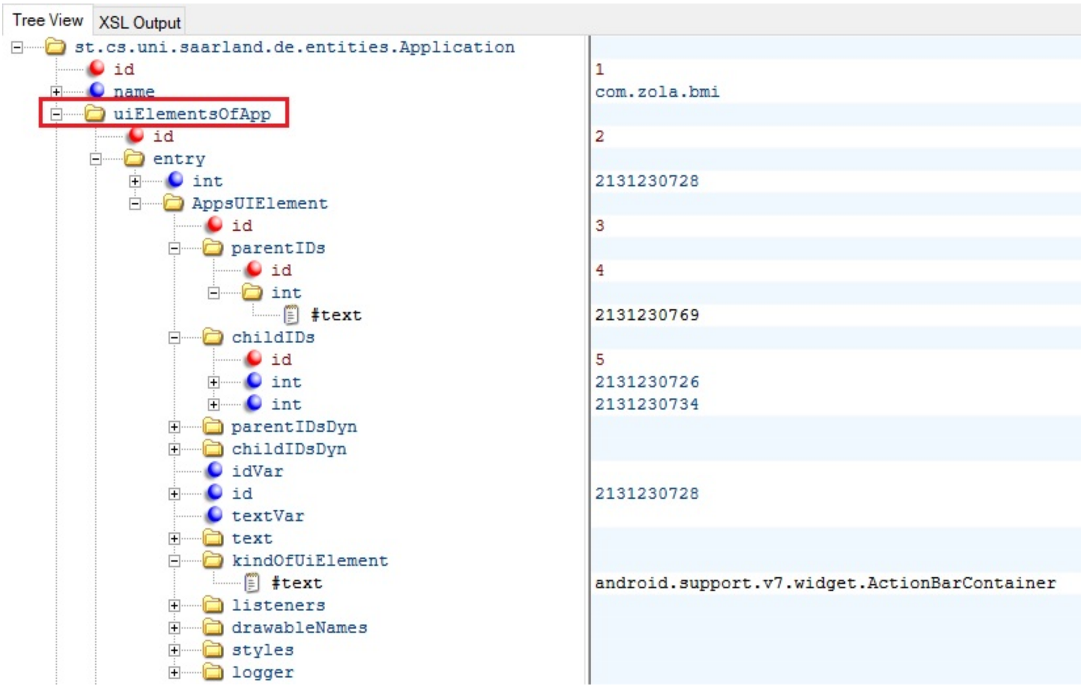

Figure 15: Details for one entry in uiElementsOfApp tag in appSerialized.txt file

```
xmlLayoutFiles - Notepad
File Edit Format View Help
XMLLayoutFile: abc_alert_dialog_title_material; 2131361802
UIElement id: 2131230747
UIElement id: 16908294
UIElement id: 2131230850
UIElement id: 2131230851
UIElement id: 2131230853

XMLLayoutFile: abc_cascading_menu_item_layout; 2131361803
UIElement id: 2131230780
UIElement id: 1000011
UIElement id: 2131230828
UIElement id: 1000010
UIElement id: 2131230765
UIElement id: 2131230849
UIElement id: 2131230838

XMLLayoutFile: abc_alert_dialog_button_bar_material; 2131361800
UIElement id: 2131230832
UIElement id: 16908314
UIElement id: 1000006
UIElement id: 16908315
UIElement id: 2131230754
UIElement id: 16908313

XMLLayoutFile: abc_alert_dialog_material; 2131361801
UIElement id: 2131230766
UIElement id: 1000009
UIElement id: 1000008
UIElement id: 2131230767
UIElement id: 1000007
UIElement id: 2131230816
UIElement id: 16908299
UIElement id: 2131230846
UIElement id: 2131230814
UIElement id: 2131230815
UIElement id: 2131230847
UIElement id: 2131230768
UIElement id: 2131230805

XMLLayoutFile: abc_list_menu_item_checkbox; 2131361806
UIElement id: 2131230759

XMLLayoutFile: abc_list_menu_item_icon; 2131361807
UIElement id: 2131230785

XMLLayoutFile: abc_dialog_title_material; 2131361804
UIElement id: 1000013
UIElement id: 1000012
UIElement id: 2131230849

XMLLayoutFile: abc_expanded_menu_layout; 2131361805
UIElement id: 2131230775

XMLLayoutFile: abc_action_menu_item_layout; 2131361794
UIElement id: 1000002
```

Figure 16: Ids of GUI elements for each resource layout file seen in xmlLayoutFiles.txt file

We ran Backstage on a small sample of 213 Android apk files. These files were picked randomly across almost 3 million apk files such that these 213 files covered a range of apk file sizes. The 3 million apk files were obtained from the repo at the University of Luxemborg who obtained these apps from the Google app store. From this collection of 3 million apps, first a sample of 1,774 apk files was randomly obtained. This set of 1,774 apk files were of varying sizes from 14 KB - 279 MB. Since we wanted a sample that covered this wide range of apk file sizes, we randomly obtained files of sizes that were in ranges like 14 KB-459 KB, 5 MB-15 MB, 39 MB-50 MB, 51 MB-87 MB and 100 MB-279 MB such that:

- 100 apk files were of size in the range of 14 KB-459 KB
- 49 apk files were of size in the range of 5 MB-15 MB
- 25 apk files were of size in the range 39 MB-50 MB
- 35 apk files were of size in the range 51 MB-87 MB
- 4 apk files were of size in the range 100 MB-279 MB

The last category has only 4 apk files and that is because the sample of 1,774 files that was randomly obtained also had only 4 Apk files. Here, the sizes refer to the size of the APK files. We only considered the Apk file size for selecting the small sample of files for testing Backstage. This was done so that we have a sample that is representative of the large collection of 3 million Apk files that we have to finally analyze. No other features of the apps were considered while selecting the sample.

Using the sample of 213 apk files, we built the pipeline in Scala with the below steps:

1. A list of APK files is created from the collection of files in the input folder.
2. Using Scala's .par operator for parallel processing of the list of APK files, a script consisting of below steps is run for each apk file:
   (a) Running the Apktool command to extract the resource and layout files of an app.
   (b) Running the Backstage command to perform static analysis
   (c) If backstage analysis is successful, mark the Apk file as done.
   (d) Errors or Success messages are appropriately logged for further reference. In case of an error with processing of any file, the program does not stop execution but rather continues with processing of the other APK files.

Table I provides summary of results seen on processing Backstage command. We see that out of 213 apk files, Backstage produces results for 134 apk files as highlighted in the last row of Table I. This accounts for about 63% of the input apps in this sample. Among the errors, we see about 32 apps fail with error "Non english language detected. Aborting", 44 apps fail with error "App has less than 70% of XML Layouts" and 3 apps fail with general Java Runtime Exceptions.

Input APK File Size	14KB-459KB	5MB-15MB	39MB-50MB	51MB-87MB	100MB-279MB	Total
#Input APKs processed	100	49	25	35	4	213
#Error-Non english language detected. Aborting	21	5	2	2	2	32
#Error-App has less than 70% of XML Layouts	16	9	8	10	1	44
#Error-Java Runtime Exceptions	1	0	0	2	0	3
#Backstage results generated successfully	**62**	**35**	**15**	**21**	**1**	**134**

TABLE I: Table showing results of Backstage run on 213 Android Apk files over a range of Apk file sizes

Thus, of the 213 apk files, we see that there are two major reasons why Backstage did not produce GUI app models:

- **Error 1: Non english language detected. Aborting**

  This is a restriction placed by Backstage in its implementation that it removes apps from consideration for which less than 70% of the resources are in the English language. This is done because Backstage tries to identify anomalies in the expected behavior of GUI objects by reading the content of labels of GUI elements. The content is needed so that Backstage can further analyze what the expected behavior should be. For example, a button with text ‘Login’ will indicate that the user wants to only login and not provide any additional sensitive information like the user’s financial details. For this analysis, Backstage works on apps with interfaces written in the English language. As the apk files have been downloaded by crawling the internet with no restrictions placed on the language, we have apps in our datastore that could belong to non-English language.

- **Error 2: App has less than 70% of XML Layouts**

  This is a second restriction placed by Backstage in its implementation to remove apps which have almost little to no GUI. It determines this based on calculating the value: (number of layout files / number of activities) <70%. "The intuition behind this heuristic is that there is usually one layout file for each activity. When this is not the case, it means that the activity does not have a UI that can be analyzed using Backstage (i.e. there are no labels with text, buttons, etc.). This heuristic, in essence led to ignoring all apps listed in the GAMES, ANDROID_WEAR, COMICS, and APP_WIDGET categories." [187]

5.3 Dynamic Analysis - AndroidRipper

As a workaround towards dealing with apps which could not be statically analyzed by Backstage, we propose to perform dynamic analysis using the AndroidRipper tool. We show below a couple of apps that worked using AndrodiRipper that had earlier failed via Backstage.

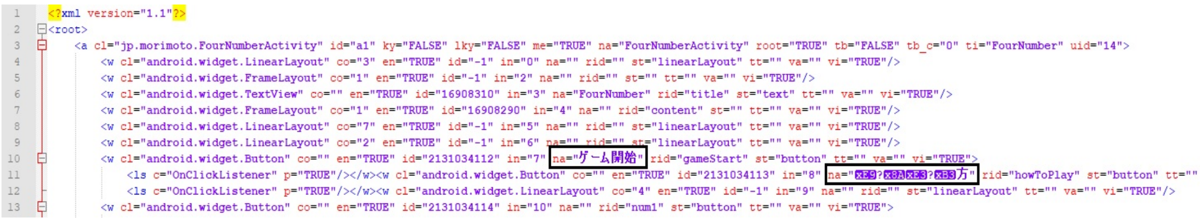

Figure 17: Part of the GUI model generated by AndroidRipper tool for an app that failed with Backstage due to error 'Non english language detected. Aborting'

To run AndroidRipper, we downloaded the latest release of the source code [197] which contains the AndroidRipper.jar file. We first run this tool on an APK file which failed while running Backstage with the error 'Non english language detected. Aborting'.

We run the tool using the command:

"java -jar AndroidRipper.jar sample.apk default.properties" where,

- **Sample.apk** is sample apk file
- **Default.properties** provides configuration values like the target device, exploration strategy

Figure 17 shows a part of the GUI model generated on one of the runs of the tool on the apk file which failed with Backstage due to non-English characters. As seen in the Figure 17, the highlighted parts in black show the non-English characters.


```xml
<?xml version="1.1"?>
<root>
<a cl="cinema.release.dates.CinemaReleaseDatesUKActivity" id="a1" ky="FALSE" lky="FALSE" me="TRUE" na="CinemaReleaseDatesUKActivity" root="TRUE"
    <w cl="android.widget.LinearLayout" co="2" en="TRUE" id="-1" in="0" na="" rid="" st="linearLayout" tt="" va="" vi="TRUE"/>
    <w cl="android.widget.FrameLayout" co="1" en="TRUE" id="16908290" in="2" na="" rid="content" st="" tt="" va="" vi="TRUE"/>
    <w cl="android.widget.RelativeLayout" co="2" en="TRUE" id="-1" in="3" na="" rid="" st="relativeLayout" tt="" va="" vi="TRUE"/>
    <w cl="android.widget.ScrollView" co="1" en="TRUE" id="2131099648" in="4" na="" rid="ScrollView02" st="" tt="" va="" vi="TRUE"/>
    <w cl="android.widget.LinearLayout" co="154" en="TRUE" id="2131099649" in="5" na="" rid="mainpage" st="linearLayout" tt="" va="" vi="TRUE"/>
    <w cl="android.widget.TextView" co="" en="TRUE" id="-1" in="6" na="March 2012" rid="" st="text" tt="" va="" vi="TRUE"/>
    <w cl="android.widget.TextView" co="" en="TRUE" id="-1" in="7" na="Friday 2nd" rid="" st="text" tt="" va="" vi="TRUE"/>
    <w cl="android.widget.TextView" co="" en="TRUE" id="-1" in="8" na="Bei Ami" rid="" st="text" tt="" va="" vi="TRUE"/>
    <w cl="android.widget.TextView" co="" en="TRUE" id="-1" in="9" na="Caranchio" rid="" st="text" tt="" va="" vi="TRUE"/>
    <w cl="android.widget.TextView" co="" en="TRUE" id="-1" in="10" na="Gone (2012)" rid="" st="text" tt="" va="" vi="TRUE"/>
```


Figure 18: Part of the GUI model generated by AndroidRipper tool for an app that failed with Backstage due to error 'App has less than 70% of XML Layouts'

Next, we ran the tool AndroidRipper on another apk file which failed on running with Backstage with error **'App has less than 70% of XML Layouts'**. Figure 18 shows part of the model extracted by the tool for one of the runs.

Thus, we see that Backstage and AndroidRipper together can help us extract the GUI model of an app.

CHAPTER 6

CONCLUSION

In this thesis, we explored existing tools available for testing Android apps that help in navigating through the screens of an app. There are mainly two types of analysis – Static and Dynamic that employ different strategies in testing an app. Of all the tools, we used Backstage that performs static analysis of the GUI layouts to help build a model of the app. We tested this on base set of 200+ Android apps. The apps for which Backstage did not produce an output model failed majorly for reasons like non-English language and apps having less than 70% of XML layouts. For such apps, we propose to use AndroidRipper which automatically performs dynamic analysis and helps in building the model. Thus, using state-of-the-art tools, we provide a framework that analyzes GUI elements, screens and their transitions across an Android app. Using these GUI models, we investigate the layouts of Android apps by collecting statistics on various GUI elements and screens.

The statistics collected in this work on existing Android apps will further help in effective simulation of SEAPHISH to determine the circumstances under which an attack against a specific app can be performed with a high degree of probability. Using phishing, i.e. a method of deception to deceive a malicious app from leaking sensitive data or from performing unauthorized activities can help defend against malware and reduce the security vulnerabilities faced by users with disabilities.

APPENDIX

I have been working as a Graduate Research Assistant on this thesis under the guidance of Dr. Mark Grechanik. The content in the sections: *Overview of App GUI Framework* and *Overview of GAP State Machine* of Chapter 1 has been referenced from Dr. Mark Grechanik's previously published work [1–8] for which Dr. Grechanik has authorized me to use the content in my thesis. The content in Chapter 2 that includes the sections: *Attack by exploiting Assistive Technologies* and *SEAPHISH: Defense by deception* has been referenced from Dr. Grechanik's previous unpublished NSF proposal. Dr. Grechanik has authorized me to use all the materials from his unpublished NSF proposal for my thesis. The *Chapter 4: Related Work* consisting of the sections: *4.1 Dynamic Analysis, 4.2 Static Analysis and 4.3 Hybrid Analysis* is written using content from a variety of research papers that helped in this thesis work.

The text in all sections (1.1, 1.2, 4.1, 4.2, 4.3) have been included in this thesis as per the below copyright policies:

- **IEEE Copyright Policy:**

  https://www.ieee.org/publications/rights/copyright-policy.html

- **ACM Copyright Policy:**

  https://www.acm.org/publications/policies/copyright-policy

- **Springer Copyright Policy:** https://www.springer.com/us/open-access/publication-policies

CITED LITERATURE

1. Grechanik, M., Mao, C. W., Baisal, A., Rosenblum, D., and Hossain, B. M. M.: Differencing graphical user interfaces. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)* , pages 203–214, July 2018.

2. Grechanik, M., Conroy, K. M., and Swaminathan, K. S.: Creating web services from gui-based applications. In *Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications* , SOCA '07, pages 72–79, Washington, DC, USA, 2007. IEEE Computer Society.

3. Conroy, K. M., Grechanik, M., Hellige, M., Liongosari, E. S., and Xie, Q.: Automatic test generation from gui applications for testing web services. In *2007 IEEE International Conference on Software Maintenance* , pages 345–354, Oct 2007.

4. Grechanik, M. and Conroy, K. M.: Composing integrated systems using gui-based applications and web services. In *IEEE International Conference on Services Computing (SCC 2007)* , pages 68–75, July 2007.

5. URL: Creating GUI Testing Tools Using Accessibility Technologies. `https://www.cs.uic.edu/~drmark/index_htm_files/TestBeds2009.pdf`, September 2019.

6. URL: Automatic Test Generation from GUI-Based Applications for Testing Web Services. `https://www.cs.uic.edu/~drmark/index_htm_files/Smart.pdf`, September 2019.

7. URL: Creating Web Services from GUI-Based Applications. `https://www.cs.uic.edu/~drmark/index_htm_files/Gaps2Ws.pdf`, September 2019.

8. URL: Composing Integrated Systems Using GUI-Based Applications and Web Services. `https://www.cs.uic.edu/~drmark/index_htm_files/Coins.pdf`, September 2019.

9. URL: User Interface. `https://stuff.mit.edu/afs/sipb/project/android/docs/guide/topics/ui/overview.html`, October 2019.

10. Amalfitano, D., Fasolino, A. R., and Tramontana, P.: A gui crawling-based technique for android mobile application testing. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops* , ICSTW '11, pages 252–261, Washington, DC, USA, 2011. IEEE Computer Society.

11. J.Foliot.: User statistics – people with disabilities. http://john.foliot.ca/user-statistics-people-with-disabilities, 2015.

12. Lopes, R., Van Isacker, K., and Carriço, L.: Redefining assumptions: Accessibility and its stakeholders. In *Proceedings of the 12th International Conference on Computers Helping People with Special Needs: Part I* , ICCHP'10, pages 561–568, Berlin, Heidelberg, 2010. Springer-Verlag.

13. Mankoff, J., Hayes, G. R., and Kasnitz, D.: Disability studies as a source of critical inquiry for the field of assistive technology. In *Proceedings of the 12th International ACM SIGACCESS Conference on Computers and Accessibility* , ASSETS '10, pages 3–10, New York, NY, USA, 2010. ACM.

14. Microsoft and Forrester: *The wide range of abilities and its impact on computer technology: A research study commissioned by microsoft corporation and conducted by forrester research, inc.,.* . Forrester Research, 2003.

15. URL: 10 years of growth of Mobile App Market. https://www.knowband.com/blog/mobile-app/growth-of-mobile-app-market, September 2019.

16. URL: Write your own Android App. http://docplayer.net/35790530-Write-your-own-android-app.html, October 2019.

17. URL: The Mobile Operating Systems That Matter in 2019. https://learn.g2.com/mobile-operating-systems, September 2019.

18. URL: Mobile Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide, September 2019.

19. URL: Category: MIUI. https://digitalspheretech.com/category/miui/page/4, September 2019.

20. URL: Google Play Installer. http://lescriva.com/ckpur8x/d9khlv.php?an=google-play-installer, September 2019.

21. URL: Google Play. `http://yourmovieinfos.blogspot.com/2017/12/google-play.html`, September 2019.

22. Michail, A.: Browsing and searching source code of applications written using a gui framework. In *Proceedings of the 24th International Conference on Software Engineering* , ICSE '02, pages 327–337, New York, NY, USA, 2002. ACM.

23. Qing Xie, Mark Grechanik, and Chen Fu: Rest: A tool for reducing effort in script-based testing. In *2008 IEEE International Conference on Software Maintenance* , pages 468–469, Sep. 2008.

24. Fu, C., Grechanik, M., and Xie, Q.: Inferring types of references to gui objects in test scripts. In *2009 International Conference on Software Testing Verification and Validation* , pages 1–10, April 2009.

25. Grechanik, M., Batory, D. S., and Perry, D. E.: Integrating and reusing gui-driven applications. In *Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools* , ICSR-7, pages 1–16, Berlin, Heidelberg, 2002. Springer-Verlag.

26. URL: Build a simple user interface. `https://developer.android.com/training/basics/firstapp/building-ui`, October 2019.

27. URL: View. `https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/view/View.html`, October 2019.

28. URL: ViewGroup. `https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/view/ViewGroup.html`, October 2019.

29. URL: Application Fundamentals. `https://developer.android.com/guide/components/fundamentals.html`, October 2019.

30. URL: Intents and Intent Filters. `https://developer.android.com/guide/components/intents-filters`, October 2019.

31. URL: Configure your build. `https://developer.android.com/studio/build`, October 2019.

32. URL: A tool for reverse engineering Android apk files. `https://ibotpeaches.github.io/Apktool`, October 2019.

33. URL: How to extract XML code from apk file in android. https://www.techwalls.com/extract-xml-code-apk-file-android, October 2019.

34. URL: Accessibility Service. https://developer.android.com/reference/android/accessibilityservice/AccessibilityService, October 2019.

35. URL: Unusual ways of using Android accessibility services. https://www.tooploox.com/blog/unusual-ways-using-android-accessibility-services, October 2019.

36. Grechanik, M., Xie, Q., and Fu, C.: Creating gui testing tools using accessibility technologies. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops* , ICSTW '09, pages 243–250, Washington, DC, USA, 2009. IEEE Computer Society.

37. Grechanik, M., Xie, Q., and Fu, C.: Maintaining and evolving gui-directed test scripts. In *Proceedings of the 31st International Conference on Software Engineering* , ICSE '09, pages 408–418, Washington, DC, USA, 2009. IEEE Computer Society.

38. Jang, Y., Song, C., Chung, S. P., Wang, T., and Lee, W.: A11y attacks: Exploiting accessibility in operating systems. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* , CCS '14, pages 103–115, New York, NY, USA, 2014. ACM.

39. Kane, S. K., Jayant, C., Wobbrock, J. O., and Ladner, R. E.: Freedom to roam: A study of mobile device adoption and accessibility for people with visual and motor disabilities. In *Proceedings of the 11th International ACM SIGACCESS Conference on Computers and Accessibility* , Assets '09, pages 115–122, New York, NY, USA, 2009. ACM.

40. PoPs. Pops ringtons notifications. https://play.google.com/store/apps/details?id=com.pops.app, September 2015.

41. G.L. D. Team. Go launcher ex notification. https://play.google.com/store/apps/details?id=com.gau.golauncherex.notification/, September 2015.

42. de Santana, V. F., de Oliveira, R., Almeida, L. D. A., and Baranauskas, M. C. C.: Web accessibility and people with dyslexia: A survey on techniques and guidelines. In *Proceedings of the International Cross-Disciplinary Conference on Web Accessibility* , W4A '12, pages 35:1–35:9, New York, NY, USA, 2012. ACM.

43. Ding, C., Wald, M., and Wills, G.: A survey of open accessibility data. In *Proceedings of the 11th Web for All Conference* , W4A '14, pages 37:1–37:4, New York, NY, USA, 2014. ACM.

44. Dorigo, M., Harriehausen, B., Stengel, I., and Haskell-Dowland, P.: Survey: Improving document accessibility from the blind and visually impaired user's point of view. volume 6768, pages 129–135, 07 2011.

45. Freire, A. P., Russo, C. M., and Fortes, R. P. M.: A survey on the accessibility awareness of people involved in web development projects in brazil. In *Proceedings of the 2008 International Cross-disciplinary Conference on Web Accessibility (W4A)* , W4A '08, pages 87–96, New York, NY, USA, 2008. ACM.

46. Luque Centeno, V., Delgado Kloos, C., Arias Fisteus, J., and Álvarez Álvarez, L.: Web accessibility evaluation tools: A survey and some improvements. *Electron. Notes Theor. Comput. Sci.* , 157(2):87–100, May 2006.

47. McMurrough, C., Ferdous, S., Papangelis, A., Boisselle, A., and Heracleia, F. M.: A survey of assistive devices for cerebral palsy patients. In *Proceedings of the 5th International Conference on PErvasive Technologies Related to Assistive Environments* , PETRA '12, pages 17:1–17:8, New York, NY, USA, 2012. ACM.

48. O'Brien, A. and Ruairi, R. M.: Survey of assistive technology devices and applications for aging in place. In *Proceedings of the 2009 Second International Conference on Advances in Human-Oriented and Personalized Mechanisms, Technologies, and Services* , CENTRIC '09, pages 7–12, Washington, DC, USA, 2009. IEEE Computer Society.

49. Vázquez, S. R. and Bolfing, A.: Multilingual website assessment for accessibility: A survey on current practices. In *Proceedings of the 15th International ACM SIGACCESS Conference on Computers and Accessibility* , ASSETS '13, pages 59:1–59:2, New York, NY, USA, 2013. ACM.

50. Yao, D., Qiu, Y., Du, Z., Ma, J., and Huang, H.: A survey of technology accessibility problems faced by older users in china. In *Proceedings of the 2009 International Cross-Disciplinary Conference on Web Accessibililty (W4A)* , W4A '09, pages 16–25, New York, NY, USA, 2009. ACM.

51. Yuan, B., Folmer, E., and Harris, Jr., F. C.: Game accessibility: A survey. *Univers. Access Inf. Soc.* , 10(1):81–100, March 2011.

52. Templeton, S. J.: Security aspects of cyber-physical device safety in assistive environments. In *Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments* , PETRA '11, pages 53:1–53:8, New York, NY, USA, 2011. ACM.

53. Cabrera-Umpiérrez, M. F., Castro, A. R., Azpiroz, J., Colomer, J. B. M., Arredondo, M. T., and Cano-Moreno, J.: Developing accessible mobile phone applications: The case of a contact manager and real time text applications. In *Proceedings of the 6th International Conference on Universal Access in Human-computer Interaction: Context Diversity - Volume Part III* , UAHCI'11, pages 12–18, Berlin, Heidelberg, 2011. Springer-Verlag.

54. Chalkia, E. and Bekiaris, E.: A harmonised methodology for the components of software applications accessibility and its evaluation. In *Proceedings of the 6th International Conference on Universal Access in Human-computer Interaction: Design for All and eInclusion - Volume Part I* , UAHCI'11, pages 197–205, Berlin, Heidelberg, 2011. Springer-Verlag.

55. Cunningham., K.: *Accessibility Handbook: Making 508 Compliant Websites* . O'Reilly Media, 2012.

56. Hoffman, D. and Battle, L.: Emerging issues, solutions & challenges from the top 20 issues affecting web application accessibility. In *Proceedings of the 7th International ACM SIGACCESS Conference on Computers and Accessibility* , Assets '05, pages 208–209, New York, NY, USA, 2005. ACM.

57. J. Lazar, D. G. and Taylor., A.: *Ensuring Digital Accessibility through Process and Policy* . Morgan Kaufmann, 2015.

58. Moreno, L. and Martínez, P.: A review of accessibility requirements in elderly users' interactions with web applications. In *Proceedings of the 13th International Conference on InteracciÓN Persona-Ordenador* , INTERACCION '12, pages 47:1–47:2, New York, NY, USA, 2012. ACM.

59. Moreno, L., Martinez, P., Ruiz, B., and Iglesias, A.: Toward an equal opportunity web: Applications, standards, and tools that increase accessibility. *Computer* , 44(5):18–26, May 2011.

60. Abd El-Sattar, H. K. H.: An intelligent tutoring system for improving application accessibility of disabled learners. In *Proceedings of the 2008 Fifth International Confer-*

ence on Computer Graphics, Imaging and Visualisation , CGIV '08, pages 286–290, Washington, DC, USA, 2008. IEEE Computer Society.

61. Anthony, L., Kim, Y., and Findlater, L.: Analyzing user-generated youtube videos to understand touchscreen use by people with motor impairments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* , CHI '13, pages 1223–1232, New York, NY, USA, 2013. ACM.

62. Cerf, V. G.: Why is accessibility so hard? Commun. ACM, 55(11):7–7, November 2012.

63. Jeschke, S., Vieritz, H., and Pfeiffer, O.: Developing accessible applications with user-centered architecture. In *Seventh IEEE/ACIS International Conference on Computer and Information Science (icis 2008)* , pages 684–689, May 2008.

64. Park, K., Goh, T., and So, H.-J.: Toward accessible mobile application design: Developing mobile application accessibility guidelines for people with visual impairment. In *Proceedings of HCI Korea* , HCIK '15, pages 31–38, South Korea, 2014. Hanbit Media, Inc.

65. Vanderheiden, G. C.: Making software more accessible for people with disabilities: A white paper on the design of software application programs to increase their accessibility for people with disabilities. *SIGCAPH Comput. Phys. Handicap.* , (47):2–32, June 1993.

66. U.S. Government. Section 508 of the Rehabilitation Act. `http://www.access-board.gov/508.htm`, 1998.

67. Lazar, J. and Hochheiser, H.: Legal aspects of interface accessibility in the u.s. *Commun. ACM* , 56(12):74–80, December 2013.

68. Grace, M., Zhou, Y., Zhang, Q., Zou, S., and Jiang, X.: Riskranker: Scalable and accurate zero-day android malware detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* , MobiSys '12, pages 281–294, New York, NY, USA, 2012. ACM.

69. Yang, C., Yegneswaran, V., Porras, P., and Gu, G.: Detecting money-stealing apps in alternative android markets. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* , CCS '12, pages 1034–1036, New York, NY, USA, 2012. ACM.

70. Apple Accessibility. http://www.apple.com/accessibility/resources, 2015.

71. Google. Section 508 compliance (vpat). https://www.google.com/sites/accessibility.html, 2015.

72. Adappa, S., Agarwal, V., Goyal, S., Kumaraguru, P., and Mittal, S.: User controllable security and privacy for mobile mashups. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications* , HotMobile '11, pages 35–40, New York, NY, USA, 2011. ACM.

73. Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D.: Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* , CCS '12, pages 217–228, New York, NY, USA, 2012. ACM.

74. Backes, M., Bugiel, S., and Gerling, S.: Scippa: System-centric ipc provenance on android. In *Proceedings of the 30th Annual Computer Security Applications Conference* , ACSAC '14, pages 36–45, New York, NY, USA, 2014. ACM.

75. Barrera, D., Clark, J., McCarney, D., and van Oorschot, P. C.: Understanding and improving app installation security mechanisms through empirical analysis of android. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* , SPSM '12, pages 81–92, New York, NY, USA, 2012. ACM.

76. Chu, D., Kansal, A., Liu, J., and Zhao, F.: Mobile apps: It's time to move up to condos. May 2011.

77. Enck, W.: Defending users against smartphone apps: Techniques and future directions. In *Proceedings of the 7th International Conference on Information Systems Security* , ICISS'11, pages 49–70, Berlin, Heidelberg, 2011. Springer-Verlag.

78. Felt, A. P., Finifter, M., Chin, E., Hanna, S., and Wagner, D.: A survey of mobile malware in the wild. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* , SPSM '11, pages 3–14, New York, NY, USA, 2011. ACM.

79. Felt, A. P., Wang, H. J., Moshchuk, A., Hanna, S., and Chin, E.: Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Conference on Security* , SEC'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

80. Zhou, Y., Jiang, X., and Grace, M.: Systematic detection of capability leaks in stock android smartphones. Jan 2012.

81. Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D.: These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* , CCS '11, pages 639–652, New York, NY, USA, 2011. ACM.

82. Jung, J., Han, S., and Wetherall, D.: Short paper: Enhancing mobile application permissions with runtime feedback and constraints. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* , SPSM '12, pages 45–50, New York, NY, USA, 2012. ACM.

83. Rangwala, M., Zhang, P., Zou, X., and Li, F.: A taxonomy of privilege escalation attacks in android applications. *Int. J. Secur. Netw.* , 9(1):40–55, February 2014.

84. Santos, N., Raj, H., Saroiu, S., and Wolman, A.: Trusted language runtime (tlr): Enabling trusted applications on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications* , HotMobile '11, pages 21–26, New York, NY, USA, 2011. ACM.

85. Vidas, T., Votipka, D., and Christin, N.: All your droid are belong to us: A survey of current android attacks. In *Proceedings of the 5th USENIX Conference on Offensive Technologies* , WOOT'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

86. Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D.: Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* , CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

87. Ismail, Q., Ahmed, T., Kapadia, A., and Reiter, M. K.: Crowdsourced exploration of security configurations. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* , CHI '15, pages 467–476, New York, NY, USA, 2015. ACM.

88. Jeon, J., Micinski, K. K., Vaughan, J. A., Fogel, A., Reddy, N., Foster, J. S., and Millstein, T.: Dr. android and mr. hide: Fine-grained permissions in android applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* , SPSM '12, pages 3–14, New York, NY, USA, 2012. ACM.

89. Meng, W., Lee, W. H., Murali, S., and Krishnan, S.: Charging me and i know your secrets!: Towards juice filming attacks on smartphones. In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security* , CPSS '15, pages 89–98, New York, NY, USA, 2015. ACM.

90. Felt, A. P., Greenwood, K., and Wagner, D.: The effectiveness of application permissions. In *Proceedings of the 2Nd USENIX Conference on Web Application Development* , WebApps'11, pages 7–7, Berkeley, CA, USA, 2011. USENIX Association.

91. Felt, A. P., Ha, E., Egelman, S., Haney, A., Chin, E., and Wagner, D.: Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security* , SOUPS '12, pages 3:1–3:14, New York, NY, USA, 2012. ACM.

92. Kelley, P. G., Consolvo, S., Cranor, L. F., Jung, J., Sadeh, N., and Wetherall, D.: A conundrum of permissions: Installing applications on an android smartphone. In *Proceedings of the 16th International Conference on Financial Cryptography and Data Security* , FC'12, pages 68–79, Berlin, Heidelberg, 2012. Springer-Verlag.

93. Liu, B., Lin, J., and Sadeh, N.: Reconciling mobile app privacy and usability on smartphones: Could user privacy profiles help? In *Proceedings of the 23rd International Conference on World Wide Web* , WWW '14, pages 201–212, New York, NY, USA, 2014. ACM.

94. Sufatrio, Tan, D. J. J., Chua, T.-W., and Thing, V. L. L.: Securing android: A survey, taxonomy, and challenges. *ACM Comput. Surv.* , 47(4):58:1–58:45, May 2015.

95. Tan, J., Nguyen, K., Theodorides, M., Negrón-Arroyo, H., Thompson, C., Egelman, S., and Wagner, D.: The effect of developer-specified explanations for permission requests on smartphone user behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* , CHI '14, pages 91–100, New York, NY, USA, 2014. ACM.

96. Sahami Shirazi, A., Henze, N., Schmidt, A., Goldberg, R., Schmidt, B., and Schmauder, H.: Insights into layout patterns of mobile user interfaces by an automatic analysis of android apps. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems* , EICS '13, pages 275–284, New York, NY, USA, 2013. ACM.

97. Chen, Q. A., Qian, Z., and Mao, Z. M.: Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *Proceedings of the 23rd USENIX Conference on Security Symposium* , SEC'14, pages 1037–1052, Berkeley, CA, USA, 2014. USENIX Association.

98. Han, J., Kywe, S. M., Yan, Q., Bao, F., Deng, R., Gao, D., Li, Y., and Zhou, J.: Launching generic attacks on ios with approved third-party applications. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security* , ACNS'13, pages 272–289, Berlin, Heidelberg, 2013. Springer-Verlag.

99. C.-C. Lin, H. Li, X. y. Z. and Wang, X.: Screenmilker: How to milk your android screen for secrets. In NDSS. The Internet Society, 2014.

100. Rodrigues, A. and Guerreiro, T.: Swat: Mobile system-wide assistive technologies. In *Proceedings of the 28th International BCS Human Computer Interaction Conference on HCI 2014 - Sand, Sea and Sky - Holiday HCI* , BCS-HCI '14, pages 341–346, UK, 2014. BCS.

101. Keng, J. C. J., Wee, T. K., Jiang, L., and Balan, R. K.: The case for mobile forensics of private data leaks: Towards large-scale user-oriented privacy protection. In *Proceedings of the 4th Asia-Pacific Workshop on Systems* , APSys '13, pages 6:1–6:7, New York, NY, USA, 2013. ACM.

102. Roesner, F., Fogarty, J., and Kohno, T.: User interface toolkit mechanisms for securing interface elements. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology* , UIST '12, pages 239–250, New York, NY, USA, 2012. ACM.

103. Azenkot, S., Rector, K., Ladner, R., and Wobbrock, J.: Passchords: Secure multi-touch authentication for blind people. In *Proceedings of the 14th International ACM SIGACCESS Conference on Computers and Accessibility* , ASSETS '12, pages 159–166, New York, NY, USA, 2012. ACM.

104. Park, C. and Jeon, J.: Application of mobile communication system to improve the accessibility of the visually disabled. In *2009 International Conference on New Trends in Information and Service Science* , pages 792–796, June 2009.

105. Verykios, V. S., Bertino, E., Fovino, I. N., Provenza, L. P., Saygin, Y., and Theodoridis, Y.: State-of-the-art in privacy preserving data mining. *SIGMOD Rec.* , 33(1):50–57, March 2004.

106. Aggarwal, C. C. and Yu, P. S.: *Privacy-Preserving Data Mining: Models and Algorithms, pages 137-156* . Springer, 2008.

107. E. Carter, A. Chiu, J. E. G. S. and Stultz, B.: When does software start becoming malware? http://blogs.cisco.com/security/talos/infinity-toolkit?f_l=sd, September 2015.

108. Gorla, A., Tavecchia, I., Gross, F., and Zeller, A.: Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering* , ICSE 2014, pages 1025–1035, New York, NY, USA, 2014. ACM.

109. Huang, J., Zhang, X., Tan, L., Wang, P., and Liang, B.: Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering* , ICSE 2014, pages 1036–1046, New York, NY, USA, 2014. ACM.

110. Yang, W., Xiao, X., Pandita, R., Enck, W., and Xie, T.: Improving mobile application security via bridging user expectations and application behaviors. In *Proceedings of the 2014 Symposium and Bootcamp on the Science of Security* , HotSoS '14, pages 32:1–32:2, New York, NY, USA, 2014. ACM.

111. Evans, A. and Wobbrock, J. O.: Input observer: Measuring text entry and pointing performance from naturalistic everyday computer use. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems* , CHI EA '11, pages 1879–1884, New York, NY, USA, 2011. ACM.

112. Potharaju, R., Newell, A., Nita-Rotaru, C., and Zhang, X.: Plagiarizing smartphone applications: Attack strategies and defense techniques. In *Proceedings of the 4th International Conference on Engineering Secure Software and Systems* , ESSoS'12, pages 106–120, Berlin, Heidelberg, 2012. Springer-Verlag.

113. Hu, W., Octeau, D., McDaniel, P. D., and Liu, P.: Duet: Library integrity verification for android applications. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks* , WiSec '14, pages 141–152, New York, NY, USA, 2014. ACM.

114. Darker: Darker's enabler. http://progress-tools.x10.mx/denabler.html, September 2015.

115. Deshotels, L.: Inaudible sound as a covert channel in mobile devices. In *Proceedings of the 8th USENIX Conference on Offensive Technologies* , WOOT'14, pages 16–16, Berkeley, CA, USA, 2014. USENIX Association.

116. Jiang, X. and Zhou, Y.: *Android Malware* . Springer Briefs in Computer Science. Springer, 2013.

117. Schlegel, R., Zhang, K., Zhou, X.-y., Intwala, M., Kapadia, A., and Wang, X.: Soundcomber: A stealthy and context-aware sound trojan for smartphones. In Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS). 01 2011.

118. Laadan, O., Shu, A., and Nieh, J.: Capture: A desktop display-centric text recorder. In *Proceedings of the 14th International ACM SIGACCESS Conference on Computers and Accessibility* , ASSETS '12, pages 9–16, New York, NY, USA, 2012. ACM.

119. Templeman, R., Crandall, D., and Kapadia, A.: Placeraider: Virtual theft in physical spaces with smartphones. 09 2012.

120. Akhter, F., Buzzi, M. C., Buzzi, M., and Leporini, B.: Conceptual framework: How to engineer online trust for disabled users. In *2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology* , volume 3, pages 614–617, Sep. 2009.

121. Ahmed, T., Hoyle, R., Connelly, K., Crandall, D., and Kapadia, A.: Privacy concerns and behaviors of people with visual impairments. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* , CHI '15, pages 3523–3532, New York, NY, USA, 2015. ACM.

122. Han, X., Kheir, N., and Balzarotti, D.: Deception techniques in computer security: A research perspective. *ACM Comput. Surv.* , 51(4):80:1–80:36, July 2018.

123. URL: Honeypots: How do you know when you are inside one? `https://ro.ecu.edu.au/cgi/viewcontent.cgi?article=1027&context=adf`, October 2019.

124. Grechanik, M., Xie, Q., and Fu, C.: Maintaining and evolving gui-directed test scripts. In *2009 IEEE 31st International Conference on Software Engineering* , pages 408–418, May 2009.

125. URL: Inferring Types of References to GUI Objects in Test Scripts. https://www.cs.uic.edu/~drmark/index_htm_files/Tigor.pdf, September 2019.

126. URL: Maintaining and Evolving GUI-Directed Test Scripts. https://www.cs.uic.edu/~drmark/index_htm_files/Rest.pdf, September 2019.

127. Section 508 of the rehabilitation act. http://www.accessboard.gov/508.htm.

128. URL: A New Era. for Accessibility. http://docplayer.net/38334528-A-new-era-for-accessibility.html, September 2019.

129. Xie, Q., Grechanik, M., Fu, C., and Cumby, C.: Guide: A gui differentiator. In *2009 IEEE International Conference on Software Maintenance* , pages 395–396, Sep. 2009.

130. URL: GUIDE: A GUI Differentiator. https://www.cs.uic.edu/~drmark/index_htm_files/GUIDE_ToolDemo.pdf, September 2019.

131. Choudhary, S. R., Gorla, A., and Orso, A.: Automated test input generation for android: Are we there yet? (e). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* , ASE '15, pages 429–440, Washington, DC, USA, 2015. IEEE Computer Society.

132. Zheng, H., Li, D., Liang, B., Zeng, X., Zheng, W., Deng, Y., Lam, W., Yang, W., and Xie, T.: Automated test input generation for android: Towards getting there in an industrial case. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track* , ICSE-SEIP '17, pages 253–262, Piscataway, NJ, USA, 2017. IEEE Press.

133. URL: Automate user interface tests. https://developer.android.com/training/testing/ui-testing/, October 2019.

134. URL: Memory Leak Patterns in Android. https://android.jlelse.eu/memory-leak-patterns-in-android-4741a7fcb570, October 2019.

135. Banerjee, A., Chong, L. K., Chattopadhyay, S., and Roychoudhury, A.: Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* , FSE 2014, pages 588–598, New York, NY, USA, 2014. ACM.

136. Liu, Y., Xu, C., and Cheung, S.-C.: Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering* , ICSE 2014, pages 1013–1024, New York, NY, USA, 2014. ACM.

137. URL: Fuzz Testing. `https://www.computerhope.com/jargon/f/fuzz-testing.htm`, October 2019.

138. Godefroid, P.: Random testing for security: Blackbox vs. whitebox fuzzing. In *Proceedings of the 2Nd International Workshop on Random Testing: Co-located with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)* , RT '07, pages 1–1, New York, NY, USA, 2007. ACM.

139. URL: Fuzz Testing. `https://www.sciencedirect.com/topics/computer-science/fuzz-testing`, October 2019.

140. URL: UI/Application Exerciser Monkey. `https://developer.android.com/studio/test/monkey`, October 2019.

141. URL: MonkeyRunner. `https://developer.android.com/studio/test/monkeyrunner`, October 2019.

142. Machiry, A., Tahiliani, R., and Naik, M.: Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* , ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.

143. URL: Dynodroid: Automated testing of Smartphone Apps. `https://dynodroid.github.io/`, October 2019.

144. URL: Automatic Input Generation System for Android Apps. `https://github.com/dynodroid/dynodroid`, October 2019.

145. King, J. C.: Symbolic execution and program testing. *Commun. ACM* , 19(7):385–394, July 1976.

146. Anand, S., Naik, M., Harrold, M. J., and Yang, H.: Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* , FSE '12, pages 59:1–59:11, New York, NY, USA, 2012. ACM.

147. Sen, K., Marinov, D., and Agha, G.: Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* , ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

148. URL: Dynamic Symbolic Execution of Android Apps. `https://github.com/saswatanand/acteve`, October 2019.

149. Mao, K., Harman, M., and Jia, Y.: Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* , ISSTA 2016, pages 94–105, New York, NY, USA, 2016. ACM.

150. URL: A Prototype of Sapienz (Out-of-date and no longer supported). `https://github.com/Rhapsod/sapienz`, October 2019.

151. Song, W., Qian, X., and Huang, J.: Ehbdroid: Beyond gui testing for android applications. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering* , ASE 2017, pages 27–37, Piscataway, NJ, USA, 2017. IEEE Press.

152. Vallée-Rai, R., Lam, P., Verbrugge, C., Pominville, P., and Qian, F.: Soot (poster session): A java bytecode optimization and annotation framework. In *Addendum to the 2000 Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (Addendum)* , OOPSLA '00, pages 113–114, New York, NY, USA, 2000. ACM.

153. URL: Soot - A framework for analyzing and transforming Java and Android applications. `https://sable.github.io/soot/`, October 2019.

154. URL: Implementing an intra procedural data flow analysis in Soot. `https://github.com/Sable/soot/wiki/Implementing-an-intra-procedural-data-flow-analysis-in-Soot`, October 2019.

155. URL: Tutorials for soot. `https://github.com/Sable/soot/wiki/Tutorials`, October 2019.

156. Song, W., Qian, X., and Huang, J.: Ehbdroid: Beyond gui testing for android applications. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* , pages 27–37, Oct 2017.

157. URL: EHBDroid. `https://github.com/wsong-nj/EHBDroid`, October 2019.

158. Choi, W., Necula, G., and Sen, K.: Guided gui testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* , OOPSLA '13, pages 623–640, New York, NY, USA, 2013. ACM.

159. URL: Automated Testing Tool for Android Applications. `https://github.com/wtchoi/SwiftHand`, October 2019.

160. Hao, S., Liu, B., Nath, S., Halfond, W. G., and Govindan, R.: Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services* , MobiSys '14, pages 204–217, New York, NY, USA, 2014. ACM.

161. URL: Programmable UI-Automation Framework for Dynamic App Analysis. `https://github.com/USC-NSL/PUMA`, October 2019.

162. Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., Pu, G., Liu, Y., and Su, Z.: Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* , ESEC/FSE 2017, pages 245–256, New York, NY, USA, 2017. ACM.

163. URL: Stoat (STochastic model App Tester) - an automatic testing tool for android apps. `https://github.com/tingsu/Stoat`, October 2019.

164. URL: Automatic Android App Explorer. `http://spruce.cs.ucr.edu/a3e/`, October 2019.

165. URL: Android Automatic App Explorer Architecture. `http://spruce.cs.ucr.edu/a3e/about.html`, October 2019.

166. Azim, T. and Neamtiu, I.: Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* , OOPSLA '13, pages 641–660, New York, NY, USA, 2013. ACM.

167. URL: Automatic Android App Explorer. `https://github.com/tanzirul/a3e`, October 2019.

168. Li, Y., Yang, Z., Guo, Y., and Chen, X.: Droidbot: A lightweight ui-guided test input generator for android. In *Proceedings of the 39th International Conference on Software Engineering Companion* , ICSE-C '17, pages 23–26, Piscataway, NJ, USA, 2017. IEEE Press.

169. URL: A lightweight test input generator for Android. `https://github.com/honeynet/droidbot`, October 2019.

170. URL: Android GUI Ripper Wiki. `http://wpage.unina.it/ptramont/GUIRipperWiki.htm`, October 2019.

171. URL: The GUI Ripper. `https://www.cs.umd.edu/~atif/GUITAR-Web/gui_ripper.htm`, October 2019.

172. Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., and Memon, A. M.: Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* , ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.

173. Amalfitano, D., Fasolino, A. R., Tramontana, P., and Amatucci, N.: Considering context events in event-based testing of mobile applications. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops* , ICSTW '13, pages 126–133, Washington, DC, USA, 2013. IEEE Computer Society.

174. Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., and Imparato, G.: A toolset for gui testing of android applications. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)* , pages 650–653, Sep. 2012.

175. URL: A toolset for the automatic GUI testing of mobile Android Applications. `https://github.com/reverse-unina/AndroidRipper`, October 2019.

176. Rountev, A. and Yan, D.: Static reference analysis for gui objects in android software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* , CGO '14, pages 143:143–143:153, New York, NY, USA, 2014. ACM.

177. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.* , 49(6):259–269, June 2014.

178. URL: GATOR: Program Analysis Toolkit For Android. `http://web.cse.ohio-state.edu/presto/software/gator/`, October 2019.

179. URL: Understand the Activity Lifecycle. `https://developer.android.com/guide/components/activities/activity-lifecycle`, October 2019.

180. URL: Event Handlers. `https://developer.android.com/guide/topics/ui/ui-events#EventHandlers`, October 2019.

181. Yang, S., Zhang, H., Wu, H., Wang, Y., Yan, D., and Rountev, A.: Static window transition graphs for android (t). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* , ASE '15, pages 658–668, Washington, DC, USA, 2015. IEEE Computer Society.

182. Wu, H., Yang, S., and Rountev, A.: Static detection of energy defect patterns in android applications. In *Proceedings of the 25th International Conference on Compiler Construction* , CC 2016, pages 185–195, New York, NY, USA, 2016. ACM.

183. Zhang, H., Wu, H., and Rountev, A.: Automated test generation for detection of leaks in android applications. In *Proceedings of the 11th International Workshop on Automation of Software Test* , AST '16, pages 64–70, New York, NY, USA, 2016. ACM.

184. Wang, Y. and Rountev, A.: Profiling the responsiveness of android applications via automated resource amplification. In *Proceedings of the International Conference on Mobile Software Engineering and Systems* , MOBILESoft '16, pages 48–58, New York, NY, USA, 2016. ACM.

185. URL: Index of /presto/software/gator/downloads. `http://web.cse.ohio-state.edu/presto/software/gator/downloads/`, October 2019.

186. Kuznetsov, K., Avdiienko, V., Gorla, A., and Zeller, A.: Analyzing the user interface of android apps. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)* , pages 84–87, May 2018.

187. Avdiienko, V., Kuznetsov, K., Rommelfanger, I., Rau, A., Gorla, A., and Zeller, A.: Detecting behavior anomalies in graphical user interfaces. In *Proceedings of the 39th International Conference on Software Engineering Companion* , ICSE-C '17, pages 201–203, Piscataway, NJ, USA, 2017. IEEE Press.

188. URL: Detecting Behavior Anomalies in Graphical User Interfaces. `https://github.com/uds-se/backstage`, October 2019.

189. Bacon, D. F. and Sweeney, P. F.: Fast static analysis of c++ virtual function calls. *SIGPLAN Not.* , 31(10):324–341, October 1996.

190. S. Rasthofer, S. A. and Bodden, E.: A machine-learning approach for classifying and categorizing Android sources and sinks. In NDSS 2014: 20th Annual Symposium on Network and Distributed System Security.

191. Yang, W., Prasad, M. R., and Xie, T.: A grey-box approach for automated gui-model generation of mobile applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering* , FASE'13, pages 250–265, Berlin, Heidelberg, 2013. Springer-Verlag.

192. Linares-Vásquez, M., White, M., Bernal-Cárdenas, C., Moran, K., and Poshyvanyk, D.: Mining android app usages for generating actionable gui-based execution scenarios. In *Proceedings of the 12th Working Conference on Mining Software Repositories* , MSR '15, pages 111–122, Piscataway, NJ, USA, 2015. IEEE Press.

193. Salihu, I., Ibrahim, R., Ahmed, B. S., Zamli, K. Z., and Usman, A.: Amoga: A static-dynamic model generation strategy for mobile apps testing. *IEEE Access* , 7:17158–17173, 2019.

194. URL: User scenario testing for Android. `https://github.com/robotiumtech/robotium/wiki`, October 2019.

195. URL: Apache Maven Project. `https://maven.apache.org/what-is-maven.html`, October 2019.

196. URL: Shell Script for running Backstage. `https://github.com/uds-se/backstage/blob/master/runApp.sh`, October 2019.

197. URL: Android Ripper 2017.10. `https://github.com/reverse-unina/AndroidRipper/releases`, October 2019.

VITA

NAME:	Joylyn Alexander Lewis
EDUCATION:	B.E., Computer Engineering, Sardar Patel College of Engineering, India, 2008 M.S., Computer Science, University of Illinois at Chicago, IL, 2019
EXPERIENCE:	University of Illinois at Chicago, Chicago, IL, Graduate Research Assistant, May 2019 – Dec 2019 CNA Insurance, Application Development Intern, June 2019 – Aug 2019 University of Illinois at Chicago, Chicago, IL, Graduate Researcher, Feb 2019 – May 2019 Accenture Services Pvt Ltd, India, Team Lead, Aug 2013 – Aug 2016 Infosys Limited, India, Technology Analyst, Sep 2008 – Aug 2013