A development methodology to help build secure mobile apps

by

Joydeep Mitra

BTECH, West Bengal University of Technology, 2010

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computer Science
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2020

Abstract

Mobile apps provide various critical services, such as banking, communication, and healthcare. To this end, they have access to our personal information and have the ability to perform actions on our behalf. Hence, securing mobile apps is crucial to ensuring the privacy and safety of its users.

Recent research efforts have focused on developing solutions to help secure mobile ecosystems (i.e., app platforms, apps, and app stores), specifically in the context of detecting vulnerabilities in Android apps. Despite this attention, known vulnerabilities are often found in mobile apps, which can be exploited by malicious apps to cause harm to the user. Further, fixing vulnerabilities after developing an app has downsides in terms of time, resources, user inconvenience, and information loss. Consequently, there is scope to explore alternative approaches that will help developers construct secure mobile apps.

Since Android and the apps that run on it are most readily available and widely used, this dissertation investigates mobile app security and solutions to secure mobile apps in the context of Android apps in two ways: (1) systematically catalog vulnerabilities known to occur in Android apps in a benchmark suite with desirable characteristics called Ghera. Ghera facilitates the continuous and rigorous evaluation of Android app security analysis tools and techniques, and (2) extend existing mobile app design artifacts such as storyboards to enable a mobile app development methodology called SeMA. SeMA considers security as a first-class citizen of an app's design and shows that many known vulnerabilities can be detected and eliminated while constructing an app's storyboard. A realization of SeMA using Android Studio tooling can prevent 49 of the 60 vulnerabilities known to occur in Android apps. A usability study with ten real-world developers using the methodology shows that the methodology is likely to help reduce development time and uncover vulnerabilities in an app's design.

A development methodology to help build secure mobile apps

by

Joydeep Mitra

BTECH, West Bengal University of Technology, 2010

A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computer Science
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2020

Approved by:

Co-Major Professor
Robby

Approved by:
Co-Major Professor
Venkatesh-Prasad Ranganath

Copyright

Abstract

Mobile apps provide various critical services, such as banking, communication, and healthcare. To this end, they have access to our personal information and have the ability to perform actions on our behalf. Hence, securing mobile apps is crucial to ensuring the privacy and safety of its users.

Recent research efforts have focused on developing solutions to help secure mobile ecosystems (i.e., app platforms, apps, and app stores), specifically in the context of detecting vulnerabilities in Android apps. Despite this attention, known vulnerabilities are often found in mobile apps, which can be exploited by malicious apps to cause harm to the user. Further, fixing vulnerabilities after developing an app has downsides in terms of time, resources, user inconvenience, and information loss. Consequently, there is scope to explore alternative approaches that will help developers construct secure mobile apps.

Since Android and the apps that run on it are most readily available and widely used, this dissertation investigates mobile app security and solutions to secure mobile apps in the context of Android apps in two ways: (1) systematically catalog vulnerabilities known to occur in Android apps in a benchmark suite with desirable characteristics called Ghera. Ghera facilitates the continuous and rigorous evaluation of Android app security analysis tools and techniques, and (2) extend existing mobile app design artifacts such as storyboards to enable a mobile app development methodology called SeMA. SeMA considers security as a first-class citizen of an app's design and shows that many known vulnerabilities can be detected and eliminated while constructing an app's storyboard. A realization of SeMA using Android Studio tooling can prevent 49 of the 60 vulnerabilities known to occur in Android apps. A usability study with ten real-world developers using the methodology shows that the methodology is likely to help reduce development time and uncover vulnerabilities in an app's design.

Table of Contents

List of Figures

List of Tables

First and foremost, I wish to thank Dr. Venkatesh-Prasad Ranganath for his continuous support during my doctoral program. His advice and insightful feedback has not only helped improve my research but has also helped me grow as an independent researcher.

I would also like to thank Dr. Torben Amtoft for his valuable time and feedback. I sincerely appreciate the many conversations I have had with him to improve my work.

I wish to thank Dr. Robby for his continuous support and encouragement, without which my research would have been incomplete.

I am grateful to Dr. Michael Higgins for his assistance. His comments and suggestions were extremely beneficial for designing the usability study to evaluate my solution.

I wish to thank Dr. Jungkwun Kim for being a part of my advisory committee and for his valuable time.

Aditya Narkar and Nasik-Nafi Muhammad collaborated with me on two projects related to this research. I wish to extend my deepest gratitude to them for contributing and helping me carry out this work. It was a pleasure to work with them over these years.

I wish to thank the Department of Computer Science at Kansas State University for providing me with an environment to learn and grow as a student. I would especially like to thank the departmental staff for helping me navigate through the program seamlessly.

I wish to thank the College of Engineering at Kansas State University and the Android Security Rewards program for providing the financial resources to help me carry out this research.

Although not directly related to my research, I would like to extend my heartfelt gratitude to Dr. Christer B. Aakeröy for providing me with an opportunity to work on a tangential line of research with his students. His encouragement and support through these years have helped me overcome numerous obstacles.

I would especially like to thank my partner, Dr. Nandini Sarkar, for supporting me in every way possible and providing me with numerous opportunities to collaborate with her on exciting research projects. We have grown together on this journey.

My parents have been a pillar of strength. Without their support and encouragement, this journey would not have been possible. I will be forever indebted to them for providing me with an environment to pursue my dreams.

My closest friends, Dr. Satyasikha Chakraborty and Vishal Kamath have profoundly impacted life. Without their unique perspectives, life would have been dull and boring.

Finally, I wish to thank my late uncle Dr. Bireswar Bose for providing me with the means to pursue higher education.

Dedication

To my parents and late grandmothers – *Debjani, Subrato, Leela, and Minati*!

For their love and belief in me ...

Preface

Research carried out at Kansas State University for this dissertation has led to the following publications:

- Mitra, J., Ranganath, V. 2017. Ghera: A Repository of Android App Vulnerability Benchmarks. In Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE). Association for Computing Machinery. DOI:https://doi.org/10.1145/3127005.3127010.

- Ranganath, V., Mitra, J. Are free Android app security analysis tools effective in detecting known vulnerabilities?. Empir Software Eng 25, 178–219 (2020). DOI:https://doi.org/10.1007/s10664-019-09749-y.

- Mitra, J., Ranganath, V. and Narkar, A. "BenchPress: Analyzing Android App Vulnerability Benchmark Suites," 2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW), San Diego, CA, USA, 2019, pp. 13-18, DOI: 10.1109/ASEW.2019.00020.

- Mitra, J., Ranganath, V. "SeMA: A Design Methodology for Building Secure Android Apps," 2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW), San Diego, CA, USA, 2019, pp. 19-22, doi: 10.1109/ASEW.2019.00021.

Chapter 1

Introduction

1.1 Motivation

Since the release of Android in 2008, it has exploded on the smartphone market. Today, Android runs on more than 85% of the worlds' smartphones [1]. The official app store for Android apps has 2.9 million apps [2]. These apps have become an integral part of our lives. We use them to perform critical tasks such as banking, communication, and entertainment. Consequently, apps need access to our personal information.

Given the ubiquity of the Android ecosystem (device + platform + apps), it is crucial to ensure that the ecosystem is safe and secure. The Open Handset Alliance (OHA), a consortium of 84 companies including Google and Samsung, is primarily responsible for developing and maintaining the Android platform [3]. The consortium is well equipped to manage platform and device security, as can be seen from the decreasing number of vulnerabilities in Android (see Figure 1.1), given its access to a large pool of resources. However, the responsibility to secure apps is shared by app stores and app developers. While app stores focus on keeping malicious apps out of the ecosystem, they still enter the ecosystem due to a variety of reasons such as installation from untrusted sources, inability to detect malicious behavior in apps, and access to malicious websites. Hence, *there is a need for app developers to secure their apps such that malicious apps cannot exploit their apps to*

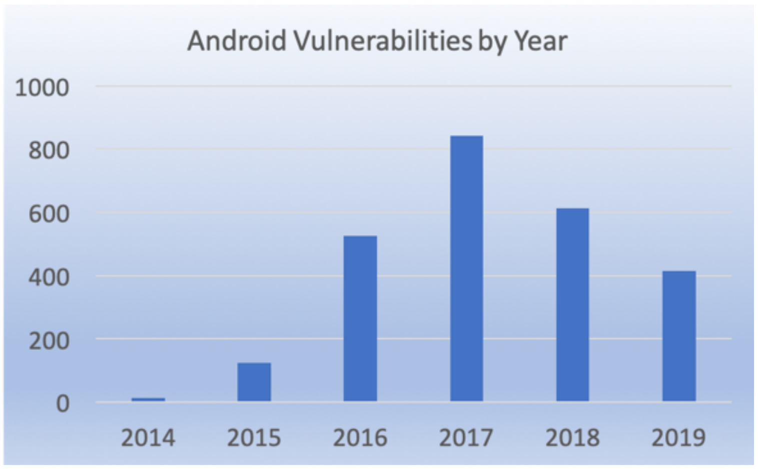


Figure 1.1: *Year-wise distribution of vulnerabilities discovered in Android*

cause harm to users.

There are approximately 724K Android app developers who contribute apps to Google Play as of January 2017 [4]. Such a large number of app developers indicate that the entry barriers to developing an Android app are low, and the incentives are high. Further, Android apps are often made by junior developers who lack awareness about security-related issues or small teams who often work in time- and resource-constrained environments [5, 6]. When developing apps, developers have to deal with numerous aspects, such as requirements elicitation, design, implementation, testing, and security. Due to the nature of app developers and app development teams, security-related aspects do not receive enough attention.

Consequently, apps are often found to be vulnerable. In 2018, an industrial study of 17 fully functional Android apps discovered that all the apps had vulnerabilities [7]. 43% of the vulnerabilities were classified as high risk, and 60% of them were on the client-side (see Figure 1.2). In 2019, a vulnerability in Google's camera app allowed a malicious app without required permissions to gain full control of the camera app and access the photos and videos stored by the camera app [8]. A recent study showed that approximately 11% of 2,000 Android apps collected from a couple of app stores, including Google Play, are vulnerable to Man-In-The-Middle (MITM) and phishing attacks [9]. Further, I helped discover a vulnerability in numerous real-world apps (e.g., the GMail app on Android) that allowed a malicious app to exploit the vulnerability and carry out phishing and denial-of-service attacks on those

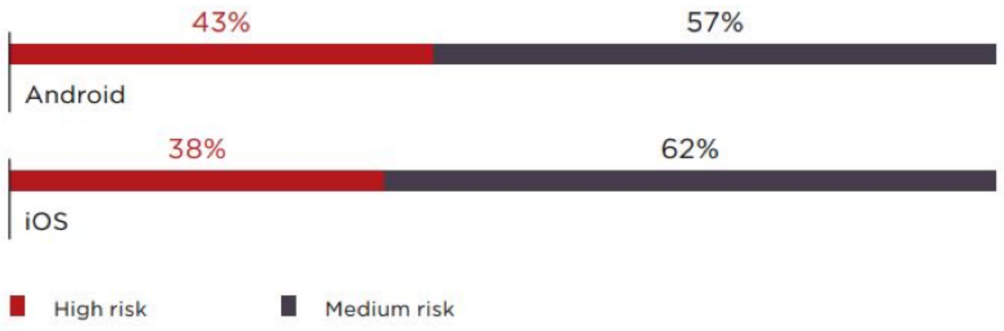


Figure 1.2: *Vulnerabilities found in Android and iOS apps in 2019*

apps [10]. The presence of high-risk vulnerabilities in Android apps suggests that app developers need help to ensure the security of their apps. Hence, in recent years, researchers have focused on developing tools and techniques to help developers identify vulnerabilities in their apps. These efforts are based on two approaches.

The first approach is to detect vulnerabilities in Android apps statically (i.e., without executing the app). For example, tools like CHEX [11], ComDroid [12], and Iccta [13] were amongst the first efforts to statically analyze Android apps to detect communication-related (ICC) vulnerabilities. Other work by Sadeghi proposed COVERT to statically analyze multiple apps to identify vulnerabilities that enable collusion attacks [14]. Another line of work related to this approach is to analyze the permissions used by an app and detect vulnerabilities related to permission over-use [15–17]. A more recent effort in this space is to use static taint analysis to track data flows that result in leaking sensitive information [18]. Tools such as FlowDroid [19] and AmanDroid [20] are the most well-known efforts that use this technique.

The second approach is to detect vulnerabilities dynamically (i.e., monitor an app during execution) [21]. Tools such as TaintDroid [22], DroidScope [23], and AppsPlayGround [24] were one of the earliest efforts in this space. They use dynamic taint analysis to monitor an app at runtime to detect data flows that lead to sensitive data leak. Most current work in securing Android apps have used static analysis techniques as opposed to dynamic analysis methods. One reason for this lopsided distribution is that performing dynamic analysis in Android is quite challenging due to Android's managed resources, binder-based inter-component communication, event triggers, and the fast-evolving Android runtime. On the other hand, static analysis of Android apps is comparatively more straightforward since Android apps are distributed as a Dalvik executable (DEX), which is similar to Java bytecode and static analysis of Java bytecode is a well-explored area of research [25].

While existing approaches to secure Android apps detect implementation bugs that cause vulnerabilities in an app, they do not help avoid or prevent flaws in an app's design that lead to security vulnerabilities in an app's implementation. As a result, many vulnerabilities are either not detected or detected after implementing an app. For example, an app storing data in the device in which it is installed is a data leak vulnerability depending on where the data is stored in the device and/or the nature of the stored data (e.g., personal identifiable information). Existing tools and techniques focused on detecting bugs in implementation are unable to detect such vulnerabilities with precision since they often lack contextual information about the app's requirements. Consequently, recent efforts have advocated for *secure by design* approaches to software development to help build secure software from the ground up [26]. *Secure by design* is a relatively recent development approach which advocates baking in security desiderata into an app's design. The design is then used as a basis for further development. In early 2014, the IEEE Computer Society Center for Secure Design emphasized the need to shift the focus in security from finding implementation bugs to identifying common software design flaws [27]. To this end, they outlined a list of the top security design flaws to help developers and designers learn from previous mistakes and avoid them during development. In spite of the growing call to integrate secure by design approach within the software development process, the existing mobile app development methodology does not enable it.

Secure by design has gained even more traction in light of recent legislation to protect personal information such as the General Data Protection Regulation (GDPR) [28] laws passed by the European Union in 2018. GDPR requires software, including mobile apps, to explicitly acquire user consent before collecting personal information. Further, it requires apps to provide users control of their data even after giving consent (e.g., right to forget).

Hence, app developers have been forced to review how they collect and manage their users' data. However, the prevalent app development approach does not encourage app developers to proactively create apps that are GDPR compliant since developers are not required to consider security properties (e.g., privacy) while designing an app [28, 29]. Consequently, app development teams with access to fewer resources struggle to comply with rules and regulations aimed at preserving user privacy [30].

In this context, the goal of this dissertation is two fold – (1) Explore and understand the vulnerabilities that occur in Android apps and (2) develop an app development methodology that bakes security into an app's design. However, before developing such a methodology this dissertation establishes its need by showing that the current prevalent approach to Android app development is not effective in securing Android apps.

1.2 Contributions

This dissertation focuses on developing a solution to construct secure mobile apps and standards to evaluate such solutions. Specifically, this dissertation makes the following contributions in the context of Android app security:

- *Systematize known Android app vulnerabilities.* Current research efforts related to Android app vulnerabilities have focused on identifying and detecting specific vulnerabilities that occur in Android apps. However, there has been no effort to systematically and comprehensively identify known Android app vulnerabilities. As a solution, *60 vulnerabilities known to plague Android apps were collected and cataloged in an informative and comprehensive repository, called Ghera, to enable rigorous and reproducible evaluation of vulnerability detection tools and techniques.*

  Ghera is publicly available at
  `https://bitbucket.org/secure-it-i/android-app-vulnerability-benchmarks/src/master/`. Ghera has helped discover vulnerabilities in Android [10, 31]. The vulnerabilities in Ghera have been used to develop tools that can be used to identify

vulnerabilities in real-world apps [32, 33]. Ghera has influenced the development of other benchmark repositories related to Android app vulnerabilities [34, 35] and other aspects such as data loss bugs [36].

In general, Ghera can be used by app developers and security researchers to understand the vulnerabilities that occur in Android apps, and as standards/benchmarks to evaluate solutions designed to help secure Android apps.

- *Characterize and assess Android app vulnerability benchmarks.* In recent years, various benchmark suites have been developed to evaluate the effectiveness of Android security analysis tools. The choice of such benchmark suites used in tool evaluations is often based on the availability and popularity of suites due to the lack of information about their characteristics and relevance. Hence, to facilitate the choice of appropriate benchmark suites for tool evaluation, *characteristics were developed to assess the various aspects of Android app vulnerability benchmarks.*

  One such aspect is representativeness of benchmarks in terms of vulnerabilities found in real-world apps. Evaluations based on representative benchmarks help determine the usefulness and relevance of the tools and techniques being measured. *This dissertation introduces a notion of representativeness based on API usage and uses it to evaluate the representativenss of benchmark suites.*

  The software artefacts used to assess the benchmark suites and required to reproduce the results of evaluating the bencmarks are publicly available at `https://bitbucket.org/secure-it-i/workspace/projects/BENCHPRESS`.

- *Develop an evaluation methodology for continuous and rigorous assessment of Android app security solutions.* Numerous tools have been developed to detect specific vulnerabilities in Android apps. However, there is no effort to comprehensively evaluate the effectiveness of these solutions to detect known Android app vulnerabilities. Therefore, *this dissertation considers 64 Android app security solutions and empirically evaluates the effectiveness of 14 Android app vulnerability detection tools in detecting the vulner-*

abilities captured in the Ghera benchmark suite.

Detailed information about the considered tools along with the raw outputs produced by the evaluated tools are publicly available in a repository to facilitate reproducibility (https://bitbucket.org/secure-it-i/evaluate-representativeness/src/master/).

- *Develop a design methodology to help construct secure mobile apps.* The current prevalent approach to mobile app security is identifying and fixing vulnerabilities after apps have been developed. This approach has downsides in terms of time, resources, user inconvenience, and information loss. As an alternative, *this dissertation proposes SeMA – a design-based mobile app development methodology aimed at preventing the creation of vulnerabilities in mobile apps.* SeMA enables app designers and developers to iteratively reason about an app's security by using its storyboard, an existing and prevalent design artifact, and generate an extensible app implementation from the storyboard. SeMA has been realized in Android Studio, the official IDE for Android app development, and is available in a publicly accessible repository (https://bitbucket.org/secure-it-i/sema/src/master/).

  While the current realization of SeMA can be used to develop Android apps, SeMA applies to software development in general since its principles are based on Model-Driven Development (MDD), a well-established software development methodology [37]. Hence, the concepts in SeMA can be extrapolated to enable the specification of domain-specific software behavior and verification of security properties using design artifacts specific to the domain.

- *Develop a formal specification of mobile app storyboards.* Mobile app storyboarding is a common design technique used to design an app's user observed behavior. SeMA extends a traditional storyboard's capabilities to facilitate the specification of an app's behavior (not specific to user observed) and reasoning about security properties at the storyboard phase of app development. This dissertation develops a formalization of the storyboard language in SeMA and the underlying security analysis. The formalization is accompanied by proofs that establish the safety and progress properties of the language's semantics, and the correctness of the analysis. Interested users of SeMA can use the formalization to understand the behavior of an app specified using the capabilities in SeMA. Further, the formalization can serve as a foundation to realize SeMA, different from the current realization, in Android or other platforms (e.g., iOS).

1.3 Organization

The dissertation is organized as follows. Chapter 2 describes a repository of Android app vulnerability benchmarks along with the design choices that went into developing the benchmarks. This chapter also defines the characteristics of vulnerability benchmarks in general and the reasons for them. Chapter 3 describes the notion of representativeness, a characteristic of vulnerability benchmarks, and explains why it is necessary for effective evaluations based on the benchmarks. It further uses this notion to measure the representativeness of four benchmark suites widely used to evaluate Android app security tools and techniques. Chapter 4 explores the existing solutions and approaches in the space of vulnerability detection in Android apps by empirically evaluating the effectiveness of 14 state-of-the-art tools in detecting the vulnerabilities in Ghera. Based on the results, this chapter argues for an alternative approach to securing Android apps. Chapter 5 describes a design methodology, called SeMA, to help develop secure mobile apps, with emphasis on Android apps. Furthermore, this chapter also presents the results of a feasibility and usability study conducted to evaluate SeMA. The final chapter summarizes the dissertation along with future directions of research.

Chapter 2

Ghera: A Repository of Android App Vulnerability Benchmarks

2.1 Motivation

The current approach to secure Android apps is to develop tools that can be used to detect vulnerabilities that can be exploited to carry out malicious actions such as data theft [11, 12, 20, 22]. However, this approach is useful only if developers trust the verdicts of the tools. One way to ensure the effectiveness of such tools is to evaluate them against a common baseline in a rigorous and reproducible manner. Rigorous means that the verdict of a tool and the reasons for the verdict can be verified. Reproducible, in this context, means that the results of a tool evaluation can be verified by repeating the same evaluation.

A common baseline enables fair and comparable evaluation. A fair evaluation means that the evaluation is not biased towards a particular tool or technique. A comparable evaluation means that the results can be compared across tools and techniques since the common baseline controls the variation across the subjects of the evaluation.

Another approach to securing Android apps is to build awareness among developers about Android app vulnerabilities. Extensive documentation and guidelines about Android app security best practices by Google [38] and other organizations such as OWASP enable developer awareness [39, 40]. Despite the availability of such resources, vulnerabilities still occur in Android apps [7], which suggests that developers are still unaware of how to develop secure apps and the benefits of developing secure apps. One solution to improve developer awareness is to create and maintain a repository of apps with known vulnerabilities. Such a catalog will help illustrate to a developer the different ways in which vulnerabilities manifest in an app. This observation is supported by numerous studies that have identified that developers are not provided with adequate support to develop secure applications [6, 41].

Moreover, existing research efforts have focused on developing tools and techniques to uncover vulnerabilities in Android apps. However, there is no single benchmark repository that systematically catalogs known Android app vulnerabilities in a tool and technique agnostic way.

Motivated by the observations about the current state, I have developed an open Android app vulnerability benchmark suite called Ghera. Currently, Ghera has 60 benchmarks. While creating Ghera, I found little guidance for creating benchmarks. Hence, in addition to creating Ghera, I identified desirable characteristics of vulnerability benchmarks. These characteristics apply to benchmarks in the context of Android app vulnerabilities but also apply to benchmarks in general.

2.2 What is Ghera?

Ghera is an open repository of Android app vulnerability benchmarks, where each benchmark contains a unique vulnerability. Ghera was developed because of a need to capture vulnerabilities that occur in Android apps in a tool/technique agnostic, easy-to-use, well documented, and comprehensive manner.

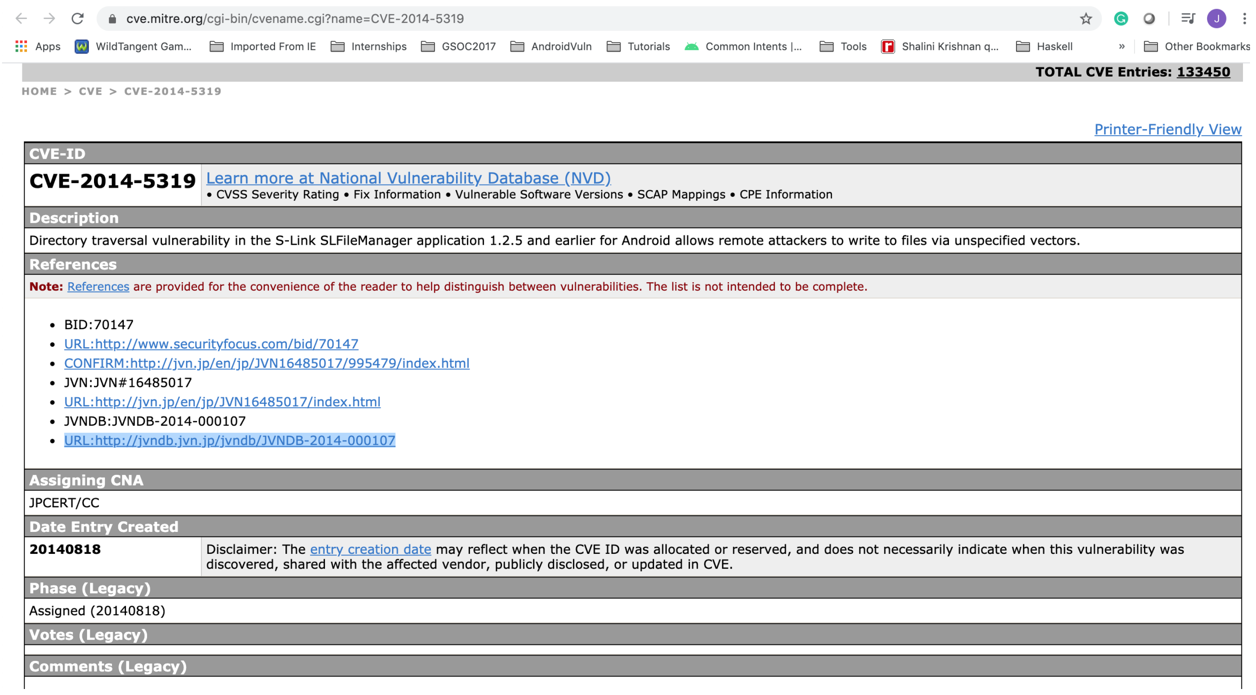
2.2.1 Design Choices

The benchmarks in Ghera are divided into categories based on the features used in the benchmarks. This categorization is useful for managing the repository. The features/capabilities used in the benchmarks were selected from the features commonly used by Android apps and discussed in Android security related resources. Almost every Android app uses one of the following:

- Perform cryptography related operations.
- Communicate with components in apps installed on the device.
- Use networking services (e.g., sockets).
- Use third-party libraries.
- Store data on and retrieve data from the device.
- Interact with the Android platform.
- Use web services.

Based on these capabilities, the benchmarks in Ghera belong to 8 categories – *Crypto, Inter-Component Communication (ICC), Networking, NonAPI, Permission, Storage, System, and Web.* In each category, I studied the relevant APIs. To identify the APIs pertinent to a potential vulnerability, I explored research efforts related to Android security, stack overflow discussions, Android source code (AOSP), and publicly disclosed vulnerability reports available at the Common Vulnerability Exposure (CVE) [42] and National Vulnerability Database (NVD) [43]. Discovering the relevant APIs from these sources is challenging due to the lack of detailed information about reproducing and exploiting the vulnerabilities related to the APIs. For example, Figure 2.1 shows a sample vulnerability report in CVE. The report provides a one line description of a complex vulnerability in an Android app with additional references. The description is not enough to understand the cause of the vulnerability, its manifestation in the app, or its exploitability. Further, the references provided do not add to the already provided information.

When I uncovered a vulnerability related to an API, I created a benign app with the API to capture the vulnerability because of the API. Further, I created a malicious app to exploit



CVE-ID	
CVE-2014-5319	Learn more at National Vulnerability Database (NVD) • CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information

Description

Directory traversal vulnerability in the S-Link SLFileManager application 1.2.5 and earlier for Android allows remote attackers to write to files via unspecified vectors.

References

Note: References are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.

- BID:70147
- URL:http://www.securityfocus.com/bid/70147
- CONFIRM:http://jvn.jp/en/jp/JVN16485017/995479/index.html
- JVN:JVN#16485017
- URL:http://jvn.jp/en/jp/JVN16485017/index.html
- JVNDB:JVNDB-2014-000107
- URL:http://jvndb.jvn.jp/jvndb/JVNDB-2014-000107

Assigning CNA

JPCERT/CC

Date Entry Created

20140818	Disclaimer: The entry creation date may reflect when the CVE ID was allocated or reserved, and does not necessarily indicate when this vulnerability was discovered, shared with the affected vendor, publicly disclosed, or updated in CVE.

Phase (Legacy)

Assigned (20140818)

Votes (Legacy)

Comments (Legacy)

Figure 2.1: *An Android App Vulnerability Report from CVE*

the vulnerability and a secure app without the vulnerability. All three apps are known as a benchmark. I verified the presence of the vulnerability in the benign app by executing the malicious app with the benign app across Android versions 4.4 - 8.1. Similarly, I verified the absence of the vulnerability in the secure app by executing the malicious app in the secure app across Android versions 4.4 - 8.1 [1]. To enable easy reproducibility, I created a test app to automatically verify the presence and absence of a vulnerability in the benchmark.

The name of each benchmark is of the form P_Q, where P is the feature that causes the vulnerability and Q is the feature that exploits the vulnerability. Based on the features used in the benchmark, the benchmark is assigned one of the eight categories mentioned before.

2.2.2 Structure and Content

The repository contains top-level folders corresponding to various categories of vulnerabilities: *Crypto, ICC, Networking, NonAPI, Permission, Storage, System, and Web.* These

[1]At the time of development Android 8.1 was the most recent version. Today, Android 6.0 - Android 10.0 are supported.

top-level folders are known as category folders. Each category folder contains subfolders corresponding to different benchmarks. These subfolders are called benchmark folders. Each category folder also contains a README file that briefly describes each benchmark in the category.

There is one-to-one correspondence between benchmark folders and benchmarks. Each benchmark folder is named as P_Q, where P is the specific feature that causes a vulnerability of interest and Q is the exploit enabled by the vulnerability. Each benchmark folder contains 4 app folders:

- *Benign* folder contains the source code of an app that uses feature P to exhibit a vulnerability,
- *Malicious* folder contains the source code of an app that exploits the vulnerability exhibited by the app in the Benign folder,
- *Secure* folder contains the source code of an app that uses feature P' to prevent the vulnerability in the Benign app, and
- *Testing* folder contains the source code of a test app that can be used to automatically verify the presence of the vulnerability in Benign and the absence of the vulnerability in Secure.

In addition to source code, each app folder has a pre-built apk distribution[2] that can run on Android versions 5.1 - 8.1.

A README file in each benchmark folder summarizes the benchmark, describes the contained vulnerability and the corresponding exploit, provides instructions to build the Benign, Malicious, and Secure apps (refer to Section 2.2.3), and lists the versions of Android on which the benchmark has been tested.

In case of Web category and some benchmarks in the Networking category, benchmark folders do not contain a Malicious folder in them because the captured vulnerabilities can be

[2]Android apps are packaged and distributed as apk files.

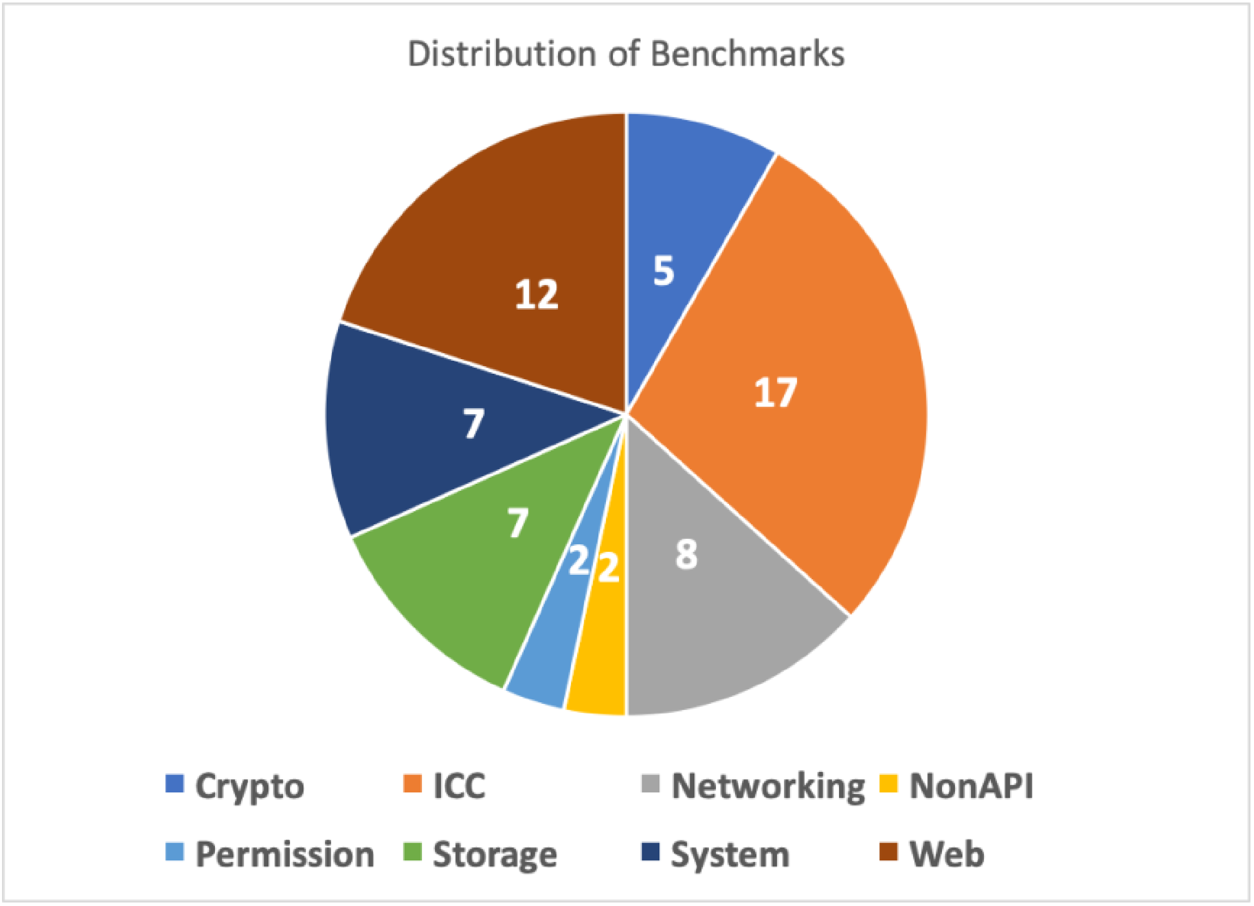


Figure 2.2: *A distribution of the benchmarks in Ghera*

exploited by Man-in-the-Middle (MitM) attacks. This requires a web server that the Benign apps can connect to. Consequently, code and instructions to set up local web server are provided in a top-level folder named Misc/LocalServer. README file of each Benign app contain instructions to configure the app to talk to the local web server. As for the MitM attack in this set up, the users are free to choose how to mount such an attack. Benchmarks in the NonAPI category rely on third party libraries. Hence, these benchmarks contain an additional folder called *Library* that contains the third-party library.

Currently, the repository contains 60 benchmarks. A distribution of the benchmarks is shown in Figure 2.2. The ICC category followed by the Web category have the highest number of benchmarks and the Permission and NonAPI categories have the least number of benchmarks.

2.2.3 Workflow

A typical workflow for using a Ghera benchmark inolves the following steps:

1. Start an emulator or device.
2. Build and install *Benign*, *Malicious*, and *Secure* apps.
3. In the emulator or device, launch the *Benign* and *Malicious* app, as per the instructions provided.
4. If the *Malicious* app successfully exploits the *Benign* app, then a message will be displayed.
5. In the emulator or device, launch the *Secure* and *Malicious* app, as per the instructions provided.
6. If the *Malicious* app fails to exploit the *Secure* app, then a message will be displayed.

A user can perform the above steps manually or use the automated test provided with the benchmark to reproduce the contained vulnerability automatically.

An example of commands needed to use a benchmark is available at: `https://bitbucket.org/secure-it-i/android-app-vulnerability-benchmarks/src/master/ICC/DynamicRegBroadcastReceiver-UnrestrictedAccess-Lean/`.

2.3 Desirable Characteristics of Vulnerability Benchmarks

2.3.1 Context

When looking for benchmarks to evaluate tools aimed at securing Android apps, I could not find any common baseline which could be used to evaluate the tools in a rigorous and reproducible manner. Most existing tool evaluations used real-world apps from app markets such as Google Play, which lacked authentic information about the presence of vulnerabilities. Some tool evaluations were based on small examples with vulnerabilities that were constructed by the tool developers. However, all the examples were geared towards a particular technique and did not explicitly capture known vulnerabilities. Hence, I started collecting and cataloging Android app vulnerabilities in an informative and open repository of benchmarks to enable fair evaluation of tools and techniques related to Android app security.

While developing the benchmarks, I searched for best practices and guidelines to create benchmarks. To my surprise, while there were numerous benchmarks, there was hardly any information about the characteristics or traits of useful benchmarks. Therefore, after collecting and cataloging 25 vulnerabilities in benchmarks, I did a retrospection and identified the characteristics of the benchmarks that I had created along with the reasons for those characteristics.

2.3.2 Vulnerability Benchmark Characteristics

This section describes the characteristics of vulnerability benchmarks that I identified along with how they were influenced by related work in the space of detecting Android app vulnerabilities.

Tool and Technique Agnostic. The benchmark is agnostic to tools and techniques and how they detect vulnerabilities. This characteristic enables the use of benchmarks for fair evaluation and comparison of tools and techniques.

Existing benchmarks capturing Android app vulnerabilities are not tool and technique agnostic. For instance, DroidBench [19], one of the first benchmark suites created in the context of efforts focused on detecting vulnerabilities in Android apps, is tailored to evaluate the effectiveness of taint analysis tools to detect information leaks in Android apps. So, the benchmarks are geared towards testing the influence of various program structures and various aspects of static analysis (e.g., field sensitivity, trade-offs in access-path lengths) on the effectiveness of taint analysis to detect information leaks (vulnerabilities). Further, it is unclear if program structures considered in the benchmarks reflect program structures that enable vulnerabilities in real world apps or program structures that push the limits of static analysis techniques.

In contrast, repositories such as AndroZoo [44] and PlayDrone [45] provide real-world Android apps available in various app stores. Further, the selection of apps is independent of their intended use by any specific tool or technique. Hence, any vulnerable apps (benchmarks) in these repositories are tool and technique agnostic.

Authentic. The benchmark provides verifiable evidence of capturing a vulnerability. Authentic benchmarks can be used as ground truths when evaluating the accuracy of tools and techniques.

To appreciate this characteristic, consider the evaluation of the tool MalloDroid [46], a tool to detect potential Man-In-The-Middle (MITM) vulnerabilities due to improper management of SSL/TLS connections (e.g., ignore SSL certificate errors). A sample of 13,500 apps was collected and analyzed using the tool to evaluate the accuracy of MalloDroid. MalloDroid reported 8% of the apps as potentially vulnerable to MITM attacks. However, the presence of a potential vulnerability does not imply that the vulnerability can be exploited. Hence, to verify MalloDroid's accuracy, 266 apps were selected from Google Play based on categories that would be most affected by the vulnerability (e.g., finance apps). Of the 266 apps, 100 apps were manually audited, and 41 of the 100 were found to have exploitable MITM vulnerabilities. This evaluation would have been simpler, easier, and more effective if authentic vulnerability benchmarks had been used.

Further, authentic benchmarks enable a fair comparison between tools. For example, consider the case when the developers of EdgeMiner [47] compared their tool's accuracy with FlowDroid [19]. They identified that EdgeMiner flagged 9 apps with vulnerabilities that were not flagged by FlowDroid. Hence, they verified this outcome by using another tool called TaintDroid. Of the 9 apps, TaintDroid flagged 4 apps. Therefore, EdgeMiner was deemed as more accurate than FlowDroid in these 4 cases. This comparison would have been simpler and more extensive if the authentic benchmarks had been used.

Existing Android app vulnerability benchmarks do not provide evidence of authenticity. For example, I discovered that 35 benchmarks in DroidBench and UBCBench [48] crash when executed on the version of Android that the benchmarks claim to target. Hence, this raises a question about the validity and authenticity of these benchmarks.

Finally, Android app repositories such as AndroZoo collect metadata and source code of real-world Android apps. While these apps can be analyzed for vulnerabilities, they do not capture vulnerabilities explicitly.

Feature Specific. If the benchmark uses only features F of a framework to create a vulnerability, then the benchmark does not contain other features of the framework that can be used to create that vulnerability. This characteristic helps evaluate if tools and techniques can detect vulnerabilities that stem only due to specific reasons (features). In other words, it helps assess if and how the cause of a vulnerability affects the ability of tools and techniques to detect the vulnerability. Often, this could translate into being able to verify the explanations provided by a tool when it detects the vulnerability.

As discussed above, EdgeMiner detected 4 more vulnerable apps than FlowDroid. However, there was no explanation for the better performance of EdgeMiner in terms of features (causes) that EdgeMiner handled better than FlowDroid. Such explanations could have been easily uncovered with feature specific benchmarks. Often, real-world apps serving as benchmarks (as in the case of AndroZoo and PlayDrone) lack this characteristic as the causes of app vulnerabilities in them are most likely unknown to the public. In contrast, benchmarks in repositories such as DroidBench exhibit this characteristic as they are feature specific by construction.

Contextual. A benchmark that captures a vulnerability in a context is different from a benchmark that captures the same vulnerability in a different context. Contextual benchmarks help evaluate the effectiveness of tools in various contexts (e.g., scale, accuracy).

Consider the size of a benchmark as an example to understand this characteristic. *Lean* benchmarks, that is, apps that use minimal features to capture a vulnerability are smaller in size than *fat* benchmarks, that is, apps that use a variety of features not specifically related to the vulnerability in the app. Lean benchmarks enable fast, easy, and effective evaluations of a tool's accuracy in detecting vulnerabilities. On the other hand, fat benchmarks can be used to evaluate the effectiveness of a tool in the context of detecting vulnerabilities at scale.

Current tool evaluation approaches use both lean and fat benchmarks. However, some tool evaluations use only fat benchmarks. As for repositories, benchmark repositories such as DroidBench and CRYPTOAPI-BENCH [34] capture vulnerabilities in lean benchmarks. In contrast, real-world app repositories such as AndroZoo and PlayDrone contain only fat benchmarks.

Representative. The benchmark captures a manifestation of a vulnerability that is similar to a manifestation of the vulnerability in real-world apps. This characteristic is useful to triage tools as per the requirements of the tool user. For example, a tool that detects vulnerabilities that are highly likely to occur in real-world apps in general might be preferred over a tool that detects vulnerabilities that are less likely to occur in real-world apps.

Previous tool evaluations have not considered the aspect of representativeness because these evaluations are based on benchmarks that do not provide evidence of or information about being representative. For example, a large number of tools use benchmark repositories such as DroidBench, ICCBench [20], and UBCBench in their evaluations. None of the repositories provide information about the representativeness of the vulnerabilities in their benchmarks. On the other hand, repositories of real-world apps such as AndroZoo are representative by design. However, such repositories do not explicitly capture vulnerable apps.

Ready-to-Use. The benchmark is composed of artifacts that can be used as is to reproduce the vulnerability. This characteristic precludes the influence of external factors (e.g., interpretation of instructions, developer skill) in realizing a benchmark. Hence, it enables fair evaluation and comparison of tools and techniques. DroidBench, AndroZoo, and PlayDrone repositories provide benchmarks as ready-to-use APKs (Android app bundles). In comparison, SEI [40] and OWASP [39] provides a set of guidelines for development of secure Android apps. The descriptions of many guidelines are accompanied by illustrative good and bad code snippets. While the code snippets are certainly helpful, they are not ready-to-use in the above sense. This is also true of many security related code snippets available as part of Android documentation.

Easy-to-Use. The benchmark is easy to set up and reproduce the vulnerability. Benchmarks with this characteristic help expedite evaluations. Consequently, this characteristic can help usher wider adoption of the benchmarks. This characteristic is desirable of benchmarks that require some assembling, e.g., build binaries from source, extensive set up after installation. As with ready-to-use characteristic, DroidBench, ICCBench, UBCBEnch, AndroZoo, and PlayDrone cater binary benchmarks that are easy to install and conduct evaluations. The source form of benchmarks provided by DroidBench also have this characteristic as they contain Eclipse project files required to build them.

Version Specific. The benchmark contains information about versions of the framework in which the captured vulnerability can be reproduced. This characteristic helps choose benchmarks when evaluating the effectiveness of tools in detecting vulnerabilities that are valid in a particular version of the framework.

To appreciate this characteristic, consider the vulnerability that allowed apps to write files to an app's shared storage space without any permission. This vulnerability was valid on Android 5.1 - 7.1 after which it was invalidated by enforcing a permission at the platform level, that is, apps needed permission to write files to another app's shared storage. If a benchmark capturing this vulnerability is not version specific and is used to evaluate a tool, then the evaluation could lead to two undesirable situations. First, the evaluation might flag the tool verdict as inaccurate if the tool correctly fails to detect the vulnerability in a version of the framework in which the vulnerability is invalid. Second, the evaluation might incorrectly flag a tool's verdict as correct even if the tool detects a vulnerability that is invalid in a version of the framework in which the tool is being evaluated.

Existing benchmark and real-world app repositories such as DroidBench, ICCBench, UBCBench, and CRYPTOAPI-BENCH are not version specific. For example, DroidBench has benchmarks which capture vulnerabilities related to reading and writing log files. Prior to Android 4.4, log files of all apps were globally accessible (i.e., an app could read/write any app's log files). However, this vulnerability is not valid after Android 4.4 because an app's log files now have restricted access. However, since DroidBench benchmarks do not have information about framework versions in which a vulnerability is valid, they could enable the undesirable situations delineated above. Further, real-world app repositories such as AndroZoo and PlayDrone do not cpature vulnerabilities explicitly. Hence, they do not have version specific information about a vulnerability.

Well Documented. The benchmark is accompanied by relevant documentation. Such documentation should contain description of the contained vulnerability and the features used to create the vulnerability. It should also mention the target (compatible) versions of the framework/platform and provide instructions to both surface the vulnerability and exploit the vulnerability. When possible, the source code of the benchmark should be included as part of the documentation. This characteristic obviously helps expedite evaluations that use the benchmarks and contributes to ease of use of benchmarks. With source code, it can help developers understand the vulnerability.

The benchmarks provided by DroidBench, ICCBench, and UBCBench are in some ways well documented as they contain source code along with binaries and there is brief documentation on the web site and in the source code about captured vulnerabilities. Benchmarks provided by CRYPTOAPI-BENCH contain source code and binaries however they do contain documentation about the vulnerabilities captured or the features used to capture the vulnerabilities. On the other hand, repositories such as AndroZoo and PlayDrone contain binaries but no source code. These repositories also contain meta-data of an app (e.g., app version, app size, and app name) but no information about vulnerabilities.

Dual. The benchmark contains both the vulnerability and a corresponding exploit (dual). This characteristic simplifies evaluations that depend on exercising the vulnerability, e.g., dynamic analysis. It allows benchmarks to be used to demonstrate vulnerabilities and even evaluate exploits. Benchmarks with this characteristic can help developers understand the vulnerability; specifically, when the source code is available. Also, duality helps verify the authenticity of benchmarks.

Existing vulnerability benchmarks do not have this characteristic.

2.3.3 Vulnerability Benchmark Repository Characteristics

Benchmark repositories are a collection of benchmarks. Similar to benchmarks, they also have desirable characteristics. I describe the characteristics that I have identified.

Open. The benchmark repository should be open to the community both in terms of consumption and contribution. The benchmarks should be available with minimal restrictions (e.g., permissive licence) and preferably at no or very low cost to the community. The repository should have a well-defined yet accessible process for the community to contribute new benchmarks. This characteristic helps with reproducibility of results and community wide consolidation of benchmarks. The latter effect reduces duplication efforts in the community.

In this regard, benchmark repositories like DroidBench, ICCBench, UCBBench, and CRYPTOPI-BENCH are more open than real-world repositories like AndroZoo and PlayDrone. Benchmark repositories are hosted publicly on the web (e.g., GitHub) and it welcomes contributions. PlayDrone is hosted as multiple public archives on Internet Archive with no explicit guidance for contributions. AndroZoo is hosted as a web service that can be accessed only by approved users. This is most likely to manage and track access to a large corpus of data in AndroZoo. Like PlayDrone, there is no explicit guidance for contributions. This may be due to how AndroZoo is populated – with real world apps collected from different app stores.

Comprehensive. The benchmark repository should have benchmarks that account for (almost) all known vulnerabilities of the target framework/platform. This characteristic simplifies evaluations as they can rely on a single repository (or very few repositories) to consider all vulnerabilities. Further, evaluations can be more thorough as they can consider most of the known vulnerabilities.

Existing repositories are not comprehensive. Each repository captures specific classes of vulnerabilities. For example, DroidBench, ICCBench, and UBCBench do a good job of covering information leak vulnerabilities due inter-component communication. On the other hand, CRYPTOAPI-BENCH captures only vulnerabilities due to the misuse of Cryptographic APIs. The lack of this characteristic makes evaluation harder since it involves using various benchmark repositories as opposed to using a comprehensive repository. For example, consider the evaluation by Reaves et al. [49], where they used DroidBench along with 6 mobile money apps and 10 most widely used financial apps in Google Play to evaluate Android security analysis tools. While DroidBench and 6 mobile money apps had certain known vulnerabilities, they did not cover all kinds of vulnerabilities. With a comprehensive repository, this evaluation could have been simpler and more thorough.

2.3.4 Characteristics of Ghera

While the desirable characteristics of vulnerability benchmarks, were identified in retrospect, they influenced the development of Ghera. Hence, in this section I will describe the characteristics that Ghera has and the reasons for those characteristics.

The vulnerabilities contained in the Ghera benchmarks were uncovered from sources related to Android app security (e.g., vulnerability reports). Further, each benchmark was verified by executing the Malicious app that exploits the Benign app and fails to exploit the Secure app. This process did not involve the use of any vulnerability (or exploit) detection tool. *Hence, the benchmarks in Ghera are tool and technique agnostic.*

Each vulnerability captured in Ghera benchmarks can be re-produced and verified automatically and manually as outlined in Section 2.2.3. *Hence the benchmarks are authentic.*

While making the benchmarks in Ghera, I used only the features required to cause the vulnerability captured in the benchmark. *Hence, the Ghera benchmarks are feature specific by construction.*

The benchmarks in Ghera focus on being minimal. Hence, they use very few features apart from the ones required to cause the vulnerability and run the app. Consequently, the benchmarks capture the vulnerability in a specific context (i.e., small-sized apps). *Therefore, the Ghera benchmarks are contextual.*

Every Ghera benchmarks come with instructions to build, install, and execute its Benign, Malicious, and Secure apps to exercise captured vulnerabilities. The manual and automated workflow to execute the benchmarks is completely verified. Hence, the work flow associated with each benchmark as illustrated in Section 2.2.3 is short, completely automated, customize-able, simple, and easy. *Hence, the benchmarks are both ready-to-use and easy-to-use.*

Each benchmark in Ghera has been tested across Android versions 5.1 - 8.1. The tests are completely automated and can be executed to reproduce the vulnerabilities across the different versions of Android supported in Ghera. *Hence, the benchmarks are version specific.*

Each benchmark is accompanied by documentation that describes the vulnerability and the associated exploit along with instructions to reproduce and exploit the vulnerability. In addition, each benchmark is available in source form. *Hence, the benchmarks are well documented.*

Each benchmark in Ghera has a Benign app that contains a unique vulnerability and a Malicious app that exploits the vulnerability in the Benign app. *Hence, the benchmarks have the dual characteristic.*

Ghera is hosted as a public repository that accepts contribution from the community. *Hence, it is an open repository.*

While Ghera does have benchmarks covering eight different areas (capabilities) of Android framework, there are many more areas of the Android framework that may be associated with known vulnerabilities and are not covered by Ghera benchmarks. *Hence, Ghera is not yet a comprehensive repository.*

Every benchmark in Ghera captures a particular manifestation of a vulnerability. There is no evidence to suggest that the manifestation of the vulnerabilities in Ghera benchmarks is similar to their manifestation in real-world apps. Hence, there is no direct evidence for Ghera being representative. However, representative benchmarks are useful for interpreting the results of an evaluation because they help assess effectiveness of the subject being evaluated in the real-world. Consequently, I have measured the representativeness of the Ghera benchmarks along with other benchmarks (e.g., DroidBench). I will discuss representativeness in detail in Chapter 3.

2.4 Limitations and Threats to Validity

Currently, Ghera has only lean benchmarks. Hence, Ghera cannot be used to evaluate the scalability of vulnerability detection tools because that would require fat benchmarks.

While Ghera currently captures 60 benchmarks across eight categories – Crypto, ICC, Networking, NonAPI, Permission, Storage, System, and Web, it does not yet contain vulnerabilities from other categories such as Bluetooth and Camera. Vulnerability reports and recently discovered vulnerabilities suggest that there might be application level vulnerabilities in those categories, which need to be explored further. This limitation can be addressed by identifying and analyzing the security-related APIs used by real-world apps but not used in the benchmarks.

Finally, the claims about the Ghera bencmarks being feature specific and contextual stems from my experience in building the Ghera benchmarks. I have tried my best to ensure that the benchmarks are minimal and use features only required to produce a particular vulnerability. However, since Ghera is open, this threat can be addressed by verifying certain metrics such as lines of code in the benchmarks or size of the apk distributions.

2.5 Conclusion

In this chapter, I have introduced an open repository of vulnerability benchmarks called Ghera. Ghera systematically catalogs 60 unique vulnerabilities discovered and collected from various sources related to Android app security. In addition to creating benchmarks, I have described the desirable characteristics of vulnerability benchmarks that were identified while developing Ghera. Further, I have argued why these characteristics are necessary. Some of the identified characteristics are harder to achieve than others. For example, creating an open benchmark repository is easier than creating a comprehensive benchmark repository. In a similar vein, creating a tool and technique agnostic benchmark is easier than achieving authenticity. However, given their relevance, benchmark developers should strive to attain these characteristics. Finally, the characteristics described in this chapter were uncovered in the context of Android app vulnerability benchmarks. However, they can be applied while creating benchmarks in general. For example, in the context of performance, benchmarks with the tool and technique agnostic characteristic can be used to evaluate and compare a variety of approaches irrespective of how they achieve performance.

Ghera is publicly available at https://bitbucket.org/secure-it-i/android-app\-vulnerability-benchmarks/src/master/.

Section A catalogs the vulnerabilities captured in Ghera.

Chapter 3

Analyzing Android App Vulnerability Benchmark Suites

In recent years, there has been a concerted effort to develop tools and techniques to help secure Android apps. One of the strategies used to evaluate such tools is to use benchmarks that contain Android app vulnerabilities [19, 20, 49]. However, the choice of benchmarks is often influenced by the popularity and availability of benchmark suites as opposed to their characteristics and relevance. One reason for this choice is the lack of information about the traits and suitability of benchmark suites related to Android app security. Hence, in this chapter, I delve deeply into one aspect of vulnerability benchmarks (i.e., representativeness). I define representativeness based on a metric called API usage and use the metric to measure the representativeness of four Android-specific benchmark suites – DroidBench [19], Ghera [50], ICC-Bench [20], and UBCBench [48]. Additionally, I verify the authenticity of the benchmarks. Finally, I explore the commonalities and differences between the benchmark suites in terms of API usage.

The findings presented in this chapter will help Android security analysis tool developers select a benchmark suite relevant to their evaluation, and benchmark developers identify gaps in their benchmark suites.

3.1 Motivation

Current Android security analysis tool evaluations use benchmarks and real-world apps for assessing their effectiveness. The effectiveness of static taint analysis tools like AmanDroid, FlowDroid, HornDroid [51], and IccTA [13] has been evaluated by applying them to benchmarks from DroidBench, ICC-Bench, and UBCBench benchmark suites and comparing tool verdicts with benchmark labels that indicate the presence/absence of specific vulnerability or malicious behavior.

Such tool evaluations have used benchmarks without considering their representativeness and authenticity. This lack of information about representativeness and authenticity has limited the results from tool evaluations in terms of the effectiveness of tools and techniques to detect vulnerabilities and their general applicability.

Recently Pauck et al. [52] developed a semi-automated technique called ReproDroid to verify the authenticity of Android app vulnerability benchmarks. They discovered that not all claims about the presence/absence of vulnerabilities in DIALDroid, DroidBench, and ICC-Bench were valid. Apart from ReproDroid and Ghera (described in Chapter 2), there has been no effort to establish the authenticity of Android app vulnerability benchmarks.

It is common in other communities to study and characterize benchmarks. In the program analysis community, Blackburn et al. [53] developed and used metrics based on static and dynamic properties of programs to characterize and compare the DaCapo benchmarks with SPEC Java benchmarks. Isen et al. [54] measured several properties of embedded Java benchmarks and how well they represent real-world mobile apps. In the systems community, Pallister et al. [55] characterized benchmarks based on the energy consumption properties of embedded platforms. In the database community, such assessments have been around since the 1990s [56]. However, such scrutiny of benchmark suites has not occurred in the Android security community.

Motivated by the above observations, I decided to evaluate Android app vulnerability benchmarks. Specifically, I intend to answer the following research questions in this chapter:

1. **RQ1** *Do Android app vulnerability benchmarks contain manifestations of vulnerabilities*

similar to real-world apps? The answer to this question will help empirically measure the representativeness of the benchmarks.

2. **RQ2** *How do the considered benchmark suites differ from each other?* The purpose of this question is to identify the common and unique features between benchmark suite pairs. The answer to this question can help tool developers choose appropriate benchmark suites to test/evaluate their tools.

3.2 Concepts and Subjects

This section defines the notion of representativeness in terms of API usage along with descriptions of the benchmarks and real-world apps used in the analysis.

3.2.1 Measuring Representativeness

A vulnerability can manifest or occur in different ways in apps due to various aspects such as producers and consumers of data, nature of data, APIs involved in handling and processing data, control/data flow paths connecting various code fragments involved in the vulnerability, and platform features involved in the vulnerability. As a simple example, consider a vulnerability that leads to information leak: sensitive data is written into an insecure location. This vulnerability can manifest in multiple ways. Specifically, at the least, each combination of different ways of writing data into a location (e.g., using different I/O APIs) and different insecure locations (e.g., insecure file, untrusted socket) can lead to a unique manifestation of the vulnerability.

Representative vulnerability benchmarks should have two aspects. First, they should capture vulnerabilities that occur in the real world. Second, the manifestation of vulnerabilities in representative benchmarks should be similar (if not identical) to that in real-world apps. Based on these aspects, there is no evidence to suggest that the vulnerability benchmarks considered here are representative. Hence, it is necessary to measure their representativeness.

One way to measure representativeness is to identify the manifestations of vulnerabilities captured in the benchmarks in real-world apps. However, this is hard since there is no definitive source of vulnerable apps. One such source is publicly available vulnerability reports (e.g., CVE [42], NVD [43], and HackerOne [57]) related to Android apps. However, these vulnerability reports have very little information about the validity, exploit-ability, and manifestation of the vulnerabilities, as discussed in Chapter 2. Further, a manual analysis of the reported apps is difficult since the reports do not make the vulnerable app versions available. While they document the version name of the vulnerable app, it is almost impossible to obtain the app for that version name since app markets such as Google Play only have the most recent version of the app. An alternative way to get the appropriate version is to download it from real-world app repositories such as AndroZoo. While such repositories *may* contain an app's version code, it is not the same as version name, which is required to download the correct version of the app.

Since the process of obtaining vulnerable apps is laborious, I decided to use usage information about Android APIs involved in manifestations of vulnerabilities as a proxy to measure the representativeness of benchmarks. *The rationale for this decision is the likelihood of a vulnerability occurring in real-world apps is directly proportional to the number of real-world apps using the Android APIs involved in the vulnerability.* So, as a weak yet general measure of representativeness, I identified the Android APIs used in the benchmarks and measured how often these APIs were used in real-world apps.

3.2.2 Considered Benchmarks

Here is a brief description of the benchmark suites that I considered in this study:

1. *DroidBench* contains 211 benchmarks. Each benchmark is an Android app that captures zero or more information leak vulnerabilities. The vulnerabilities captured in DroidBench primarily stem from the Inter-Component Communication (ICC) feature of Android and the general features of Java (e.g., dynamic dispatch).

2. *Ghera* contains 60 benchmarks. Each benchmark captures a known and unique Android

app vulnerability. Each benchmark is composed of 3 apps – *Benign*, *Malicious*, and *Secure*. Ghera is described in Chapter 2.

3. *ICC-Bench* contains 24 benchmarks. Each benchmark is an Android app that captures zero or more information leak vulnerabilities. IccBench focuses on capturing vulnerabilities that stem from communication between apps via ICC.

4. *UBCBench* contains 16 benchmarks. Each benchmark is an Android app that captures at most one information leak vulnerability. UBCBench captures information flow vulnerabilities primarily stemming from ICC and SharedPreferences [1] features of Android and general Java features (e.g., Threads).

3.2.3 Real-world Apps

I collected 700K apps from AndroZoo in March 2019. From this set of 700K apps, I curated a set of 473K apps that target API levels 19 through 27. An API level uniquely identifies the framework API revision offered by a version of the Android platform. In an Android app, the *minimum API level* is the least framework API version required by the app, and *target API level* is the framework API version targeted by the app. For this evaluation, I initially picked target API level 19 thru 27 because most benchmarks targeted these API levels. However, I later discovered that Android currently does not support API levels 19 thru 22. Therefore, to make the evaluation current, from the set of 473K apps, only apps that target API levels 23 thru 27 were retained.[2] Hence, I ended up with a sample of 226K real-world Android apps. Table 3.1 provides the distribution of this sample across considered target API Levels.

[1] A SharedPreference is a file that stores key-value pairs and can be private to an app or shared

[2] At the time of the evaluation, API level 27 was the latest. Since then, Android has released two more versions – API levels 28 and 29.

Target API level	# Real World Apps
23	146K
24	18K
25	16K
26	29K
27	17K
Total	226K

Table 3.1: *Distribution of target API levels in the sample of real world apps*

3.3 Experiment

3.3.1 Preparing the benchmarks

By design, each Android app is bundled as a self-contained APK file that contains all code and resources necessary to execute the app but are not provided by the underlying Android framework. However, as a result of the build process of Android apps, the APKs may contain unnecessary code and resources. So, ProGuard tool can be used as part of the Android app build process to remove unnecessary artifacts from APKs [58].

All the benchmark suites considered in this evaluation provide pre-built APKs and source files for each of their benchmarks. Except for Ghera, the pre-built APKs that come with each benchmark suite contain unnecessary code and resources. Further, the benchmarks do not have the same minimum and target API levels. Specifically, DroidBench benchmarks have minimum API level 8 and target API level 14 thru 24, Ghera benchmarks have minimum API level 22 and target API level 27, ICC-Bench benchmarks have minimum and target API level 25, and UBCBench benchmarks have minimum and target API level 19.

Since my objective is to measure the representativeness of benchmark suites and compare them based on API usage, I needed to control for the effects of unnecessary APIs and API level on the findings of the evaluation. Therefore, I rebuilt every benchmark from its source with minimum API level set to 23, target API level set to 27, using appcompat support library version 27.1.1, and using Proguard to remove unnecessary APIs. I chose API levels 23 thru 27 because they were currently supported by Android at the time of this evaluation.

After rebuilding the benchmarks, I ensured the benchmarks were indeed supported by

Benchmark Suite	# Total benchmarks	# Total apps	# Apps built successfully	# Apps crashed
DroidBench	211	211	201	32
Ghera	60	180	180	0
IccBench	24	24	24	0
UBCBench	16	16	16	3

Table 3.2: *Total No. of benchmarks in each benchmark suite along with the No. of benchmarks that built successfully with minimum API level 23 and target API level 27 and crashed on an emulator running Android 23 and 27.*

API levels 23 thru 27 by executing each benchmark on an emulator running Android 23 and 27. As part of the execution, I interacted with the app until no further interaction was possible. Often, this meant interacting with various widgets on a screen and navigating to various screens in an app. If the benchmark or app crashed, then I recorded the crash.

The process of re-building and testing the apps was semi-automated. Building the apps, starting an emulator for the appropriate API level, and installing the app in an emulator with the corresponding API level was automatic. Interaction with the app was manual. Table 3.2 lists the total number of benchmarks in each suite, the total number of apps in each suite, the number of apps that were built successfully, and the number of apps that crashed during execution. In this evaluation, I considered all benchmarks that could be built successfully including the ones that crashed because the reasons for a crash were unclear. A crash could have occurred due to a vulnerability intentionally captured in the benchmark or other reasons such as change of API levels.

Observations From Table 3.2, we see that, most benchmarks/apps not only build but also execute on the currently supported versions of Android even when they were not designed to run on those versions – out of 311 benchmarks across all benchmark suites, only 35 crashed during execution and only 10 could not be built successfully. So, *while most benchmarks were not explicitly designed to run on recent versions of Android, they are well supported by recent versions of Android.*

The 10 benchmarks (from DroidBench) that failed to build imply *some benchmarks are not supported by recent versions of Android.*

For the 35 benchmarks (32 from DroidBench and 3 from UBCBench) that crashed when executed on emulators running Android 23 and 27, I re-executed pre-built counterparts of these benchmarks on an emulator running the version of Android originally targeted by the benchmarks. For example, for a benchmark that was designed to target API level 14, I executed the benchmark on an emulator with API level 14. Interestingly, all of the benchmarks crashed during re-execution. *This observation raises a question about the authenticity of the benchmarks.*

None of the benchmarks in Ghera and ICC-Bench crashed. If not crashing is considered an indicator of authenticity, then the observation suggests *that the benchmarks in Ghera and ICC-Bench are more authentic than the benchmarks in DroidBench and UBCBench.* This result complements Ghera since Ghera is known to be authentic, as described in Chapter 2. However, there is no similar evidence to support that ICC-Bench is more authentic than the others.

3.3.2 API-based App Profiling

Android apps access various capabilities of the Android platform via features of XML-based manifest files and Android programming APIs. The published Android programming APIs and the XML elements and attributes (features) of manifest files are collectively referred to as APIs here. I use the term API to mean either a function, a method, a field, or an XML feature.

For each Android app, I collected the APIs used by the app or defined in the app. Of the collected APIs, I retained the *considered* APIs as follows:

1. The manifest file of an app captures the meta-data of an app in XML form. From the various elements and attributes that can be present in the manifest, I identified the value of 7 attributes and the presence of 26 attributes based on my knowledge of the benchmarks.

2. From an app's source code, I considered all published (i.e., public and protected) methods along with all methods that were used but defined in the app. The former

accounted for callback APIs provided by the app, and the latter accounted for external APIs used by the app. From these APIs, obfuscated APIs with single character names were discarded. To make apps comparable in the presence of definitions and uses of overridden Java methods (APIs), if a method was overridden, then I considered the fully qualified name (FQN) of the overridden method in place of the FQN of the overriding method using Class Hierarchy Analysis. Since I wanted to measure representativeness in terms of Android APIs, I discarded APIs whose FQN did not have any of these prefixes: `java`, `org`, `android`, and `com.android`. For each app, I recorded the remaining APIs.

3. Numerous APIs are commonly used in almost all Android apps and are related to aspects (e.g., UI rendering) that are not the focus of vulnerability benchmarks related to Android apps. I ignored such APIs while determining the APIs used in an app to ensure that these APIs do not inflate representativeness. For this purpose, I created a baseline app with minimum and target API levels set to 23 and 27, respectively. This app did not exhibit any interesting functionality but contained graphical widgets commonly used by Android apps. Out of the 1847 APIs used in this baseline app, I manually identified 1586 APIs as commonly used in Android apps; almost all of them were basic Java APIs or related to UI rendering and XML processing. For each app, I removed these APIs from its list of APIs recorded in above steps 1 and 2 and *considered* the remaining APIs.

The above steps were used to create a *relevant* API profile for the set of real-world apps and each benchmark suite.

Further, for each benchmark suite, I did additional filtering on the *considered* list of APIs obtained after removing the 1585 APIs from the baseline app. Even in the *considered* set, I identified APIs orthogonal to the focus of these benchmarks (e.g., *android.graphics*). So, I ignored these APIs to create a set of *filtered* APIs. This additional step did not change the API sets drastically, as can be seen from the 2nd and 3rd columns in Table 3.3.

Suite	Number of APIs				
	Total	Considered	Filtered	Relevant	Security
DroidBench	2188	837	798	769	744
Ghera	1906	565	518	504	494
IccBench	185	102	70	70	70
UBCBench	751	127	99	98	96

Table 3.3: *Number of APIs used by the benchmark suites*

3.3.3 Using Android app developer discussions in Stack Overflow to identify relevant and security-related APIs

The *filtered* set of APIs identified while creating the API profile is based on my notion of relevance and knowledge of the benchmarks. Since my subjectivity and bias could influence the findings of the evaluation, I used developer discussions on Stack Overflow (SoF) to recognize APIs that developers deem *relevant* to Android app development and related to *security*. The *relevant* and *security-related* APIs are listed in columns 5 and 6 in Table 3.3.

For this purpose, I used a snapshot of Stack Overflow posts from March 2019.

Identify Android related posts I considered all posts from the Stack Overflow snapshot with the *Android* tag, which resulted in 1.2 million posts. While collecting posts with the *Android* tag, I hypothesized that I was missing Android-related posts without the *Android* tag. To avoid missing such posts, I collected all tags (including synonyms) from *Android Stack Exchange*, a forum for discussing issues related to Android. This exercise resulted in 1347 tags, from which I removed 433 tags since they had been considered previously. From the remaining 914 tags, I ignored tags related to company names since they were not related to Android app development. Finally, of the remaining 445 tags, I discovered that only five tags[3] had posts associated with them on SoF. I found that these tags are not related to Android app development hence ignored them. This discovery ensured that there were no posts on SoF associated with Android app development that did not have the *Android* tag.

[3]These tags were instapaper, pebble, sdhc, task-management, and xfat

Identify Android security-related posts There is no readily available information in Stack Overflow data to identify security-related posts. So, I used the data from *Security Stack Exchange*, a forum for discussing software security issues, to identify security-related posts on SoF. I gathered all tags from Security Stack Exchange as the set of *security-related tags*. Further, I added the *security* tag to this set. From the SoF posts that had *Android* tag, I identified posts with at least one *security-related* tag. This identification resulted in a set of 460K posts related to *Android security*.

Filter posts based on API levels Since API level 23 was released in 2015, I considered only posts that had some activity, that is, created, answered, edited, commented on, voted on, or marked as favorite or accepted on or after 2015. An API that did not garner interest after 2015 is likely deprecated in API levels 23 thru 27 or well understood by developers. In either case, such APIs as irrelevant to the evaluation. This filtering resulted in 831K Android-related posts and 318K Android security-related posts.

Identify relevant and security-related APIs If the class name and the method/field name of an API co-occurred in a post, then I deemed the post as discussing the API. Based on this notion, if a filtered API was discussed in an Android-related post, then the API was considered a *relevant API* (i.e., relevant to Android app development). Similarly, if a *filtered* API was discussed in an Android security related post, then I deemed it as a *security-related API*.

3.3.4 Calculating Representativeness

For each *relevant* API used in a benchmark suite, I calculated the percentage of real-world apps using the API. If a large number of real-world apps use a large number of APIs used in a benchmark suite, then it is highly likely that real-world apps will have manifestations of vulnerabilities similar to the ones captured in the benchmarks.

In addition to *relavant* APIs, I also calculated representativeness of a benchmark suite in terms of *security-related* APIs used by benchmarks in the suite.

3.4 RQ1:Representativeness

In this section, I will use API usage to determine if benchmarks in DroidBench, Ghera, ICC-Bench, and UBCBench capture manifestations of vulnerabilities similar to the ones found in real-world apps.

For each benchmark suite, the corresponding graph in Figure 3.1 shows the percentage of real-world apps using a relevant API that is used by the suite along with the percentage of Stack Overflow posts discussing the same API.

Tables 3.4 and 3.5 list the five-number summary of the percentage of posts discussing relevant and security-related APIs, respectively.

DroidBench Of the 798 filtered APIs used by DroidBench, only 29 are not discussed by any Android related Stack Overflow post. The remaining 769 APIs are discussed in at least 1 post. As seen in Table 3.4, more than half the 769 relevant APIs used by DroidBench are discussed by at least 385 posts. Therefore, *APIs used by DroidBench are relevant as deemed by Android app developer discussion in SoF posts.*

The graph for DroidBench in Figure 3.1 shows that all relevant APIs are used by real-world apps and 562 (73%) *relevant* APIs are used by more than 60% of real-world apps. Therefore, *DroidBench is representative of real-world apps in terms of API usage.*

Ghera Of the 518 filtered APIs used by Ghera, only 14 are not discussed in any Android related Stack Overflow post. The remaining 504 (relevant) APIs are discussed in at least one *Stack Overflow* post. As seen in Table 3.4, more than half the 504 relevant APIs are discussed by at least 845 posts. Therefore, *APIs used by Ghera are considered relevant by Android app developers.*

The graph for Ghera in Figure 3.1 shows that all *relevant* APIs are used by real-world apps and 340 (67%) relevant APIs are used by more than 60% of real-world apps. Therefore, *Ghera is representative of real-world apps in terms of API usage.*

ICC-Bench All of the 70 filtered APIs used by ICC-Bench are discussed by at least one Android related Stack Overflow post. As seen in Table 3.4, more than half of the 70 relevant APIs used by ICC-Bench are discussed by at least 2446 posts. Therefore, *APIs used by ICC-Bench are highly relevant to Android app development as deemed by related discussions in StackOverflow.*

The graph for IccBench in Figure 3.1 shows that all relevant APIs are used by real-world apps and 55 (78%) relevant APIs are used by more than 60% of real-world apps. Therefore, *ICC-Bench is representative of real-world apps in terms of API usage.*

UBCBench All of the 99 filtered APIs, except 1, used by UBCBench are discussed by at least one Android related Stack Overflow post. As seen in Table 3.4, more than half the 98 relevant APIs used by UBCBench are discussed by at least 1406. Therefore, *APIs used by UBCBench are being discussed by Android app developers.*

The graph for UBCBench in Figure 3.1 shows that all relevant APIs are used by real-world apps and 79 (81%) relevant APIs are used by more than 60% of real-world apps. Therefore, *UBCBench is representative of real-world apps in terms of API usage.*

In summary, *DroidBench, Ghera, ICC-Bench, and UBCBench all use APIs that are used by a large number of real-world apps. Hence, they are all representative in terms of API usage.*

3.4.1 Discussion

Comparatively, DroidBench (769) and Ghera (504) use more than five times the number of relevant APIs used by UBCBench (98) and IccBench (70). In terms of the percentage of relevant APIs used by more than 60% of the real-world apps, DroidBench (562) and Ghera (340) use more than four times the number of relevant APIs used by UBCBench (79) and IccBench (55). So, *in terms of coverage of APIs used by real-world apps, DroidBench and Ghera fare better than UBCBench and IccBench.*

In Figure 3.1, most of the spikes in the line corresponding to Android related Stack Overflow posts are associated with relevant APIs that are used by more than 60% of the real-world apps. Hence, the benchmarks are not only representative but also relevant since they are using APIs that are not only used by a large number of real-world apps but are also being discussed widely by Android app developers.

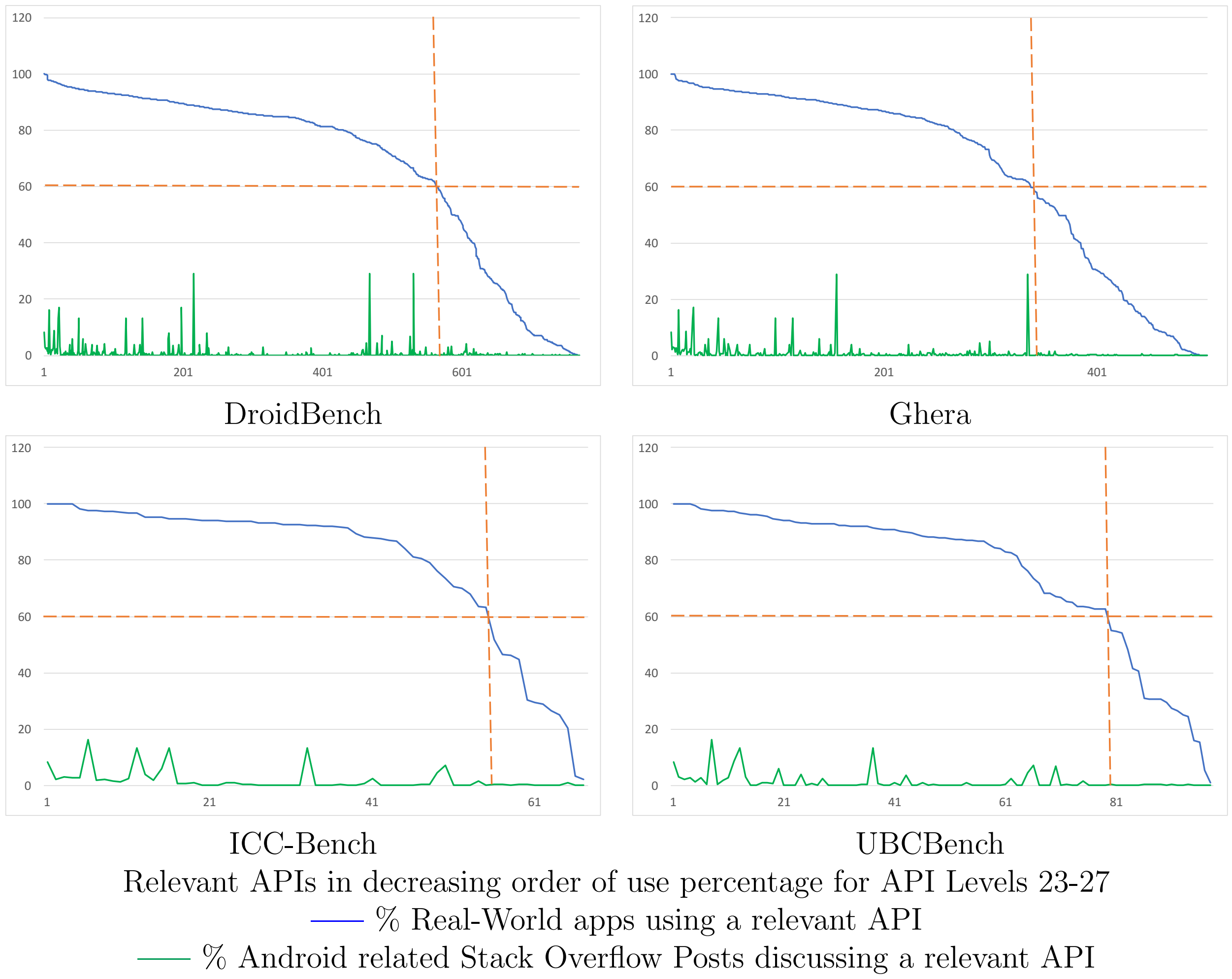


Figure 3.1: *Percentage of real-world apps that use a relevant API and the percentage of Stack Overflow posts that discuss a relevant API in a benchmark suite.*

3.4.2 What about Security-related APIs?

As shown in Table 3.3, the number of *security-related* APIs as deemed by Stack Overflow data is similar to the number of *relevant* APIs. For example, in ICC-Bench, the number of security-related APIs is identical to the number of relevant APIs. Moreover, as seen in Table 3.4, the distribution of posts discussing security-related APIs is similar to that of relevant

Repo	% (No.) of relevant posts discussing APIs									
	Min		Q1		Median		Q3		Max	
DroidBench	0.0001	(1)	0.0068	(57)	0.04	(385)	0.29	(2423)	29.0	(240K)
Ghera	0.0001	(1)	0.0156	(130)	0.10	(845)	0.44	(3668)	29.0	(240K)
ICC-Bench	0.0007	(6)	0.0588	(489)	0.29	(2446)	1.71	(14242)	15.3	(127K)
UBCBench	0.0001	(1)	0.0348	(289)	0.17	(1406)	0.95	(7880)	15.3	(127K)

Table 3.4: *Five-Number summary of 831K relevant posts discussing APIs in a benchmark suite*

Repo	% (No.) of security-related posts discussing APIs									
	Min		Q1		Median		Q3		Max	
DroidBench	0.0003	(1)	0.007	(22)	0.05	(168)	0.31	(997)	35.0	(111K)
Ghera	0.0003	(1)	0.020	(65)	0.11	(359)	0.49	(1559)	35.0	(111K)
ICC-Bench	0.0009	(3)	0.056	(179)	0.273	(869)	1.55	(4940)	14.3	(45K)
UBCBench	0.0006	(2)	0.033	(106)	0.20	(640)	1.08	(3458)	14.3	(45K)

Table 3.5: *Five-Number summary of 318K security-related posts discussing APIs in a benchmark suite*

APIs as can be seen from the five-number summary. Consequently, the observations for relevant APIs carries over to security-related APIs.

Caveat The numbers in Table 3.4 suggest that almost all the relevant APIs used by a benchmark suite are related to security. These numbers are based on the approach of identifying relevant and security-related APIs using Stack Overflow data, as explained in Section 3.3.3. While this approach considers the occurrence of APIs in posts to recognize an API as relevant or security-related, it does not consider the context in which the API occurs in posts, that is, an API can occur in a post as part of a code snippet that is being discussed but yet not be discussed in the post. Hence, the approach can conservatively identify APIs that are irrelevant or not related to security as relevant or security-related. Therefore, *the number of relevant APIs and security-related APIs used by a benchmark suite is likely lower than the numbers reported here.* This observation is supported by an earlier examination on Ghera, where I had manually identified security-related APIs used by Ghera based on my knowledge of the Ghera benchmarks. According to that study, 601 APIs had been deemed relevant to Android app development, and 117 of them were considered security-related. Interestingly, all the 117 security-related APIs are included in the set of SoF-based security-related APIs for Ghera. Further, the 117 manually curated security-related APIs are used by more than 65% real-world apps in this sample. This number strongly suggests that at least for Ghera, the conservative estimate of relevant and security-related APIs does not affect its representativeness.

3.5 RQ2: Comparison

This section examines the differences between the considered benchmark suites in terms of API usage by comparing them in a pair-wise fashion. Further, based on the identified differences, I will provide benchmark selection recommendations for tool evaluations.

For each pairwise comparison, Table 3.6 shows the number of filtered APIs common and unique to the compared benchmarks.

Benchmark Suite Pair (X/Y)	Common APIs	APIs unique to X	APIs unique to Y
DroidBench/Ghera	344	454	174
DroidBench/ICC-Bench	67	731	3
DroidBench/UBCBench	89	709	10
Ghera/ICC-Bench	42	476	28
Ghera/UBCBench	85	433	14
ICC-Bench/UBCBench	16	54	83

Table 3.6: *Filtered APIs based pairwise comparison of benchmark suites*

3.5.1 DroidBench vs Ghera

Observation 1 DroidBench uses 1.5 times the number of APIs used by Ghera but it contains almost 3 times the number of benchmarks in Ghera; see Table 3.3. Therefore, *the difference in API usage between DroidBench and Ghera is not comparable to the difference in the number of benchmarks in them.* This is most likely because DroidBench focuses on heavily ICC (depth) whereas Ghera focuses on ICC and other Android features (breadth).

Consequently, the benchmarks in DroidBench use more common APIs compared to the benchmarks in Ghera.

Observtion 2 DroidBench uses 454 APIs not used by Ghera; see Table 3.6. 438 of these APIs were identified as relevant using Stack Overflow data. Moreover, 299 (68%) relevant APIs are used by at least 60% real-world apps. Of the 438 relevant APIs, approximately 100 are not specific to Android but related to Java. Of the remaining APIs, close to 50% are related to ICC. *Since a large number of APIs unique to DroidBench are related to ICC, evaluations of Android app vulnerability detection tools, especially the ones that focus on ICC, should consider DroidBench.*

Moreover, 344 APIs are common to DroidBench and Ghera. Of these, 331 APIs are relevant and 280 (85%) APIs are used by at least 60% of real-world apps. 200 of the 344 APIs are related to ICC. This is not surprising as DroidBench focuses on ICC. Therefore, tools focusing on detecting vulnerabilities stemming from ICC can use either DroidBench or Ghera for evaluation. However, *since only 65 of the 174 APIs unique to Ghera are related to ICC, such tools should prefer DroidBench over Ghera.*

Observation 3 From Table 3.6, we see Ghera uses 174 APIs that are not used by DroidBench. 173 of these APIs were identified as relevant using Stack Overflow data. Of these relevant APIs, 69 (40%) APIs are used by at least 60% real-world apps. Further, 93 (54%) of the 173 relevant APIs are related to Android features such as web, crypto, storage, and networking features. *Since Ghera benchmarks capture vulnerabilities stemming from the use of APIs not related to ICC, evaluations of tools that detect vulnerabilities not related to ICC should consider Ghera.*

Observation 4 All benchmarks in Ghera capture vulnerabilities that can be reproduced and exploited on API Levels 23 thru 27. However, DroidBench benchmarks were designed to run on older API Levels. While I was able to build the benchmarks and install them on emulators running API Levels 23 thru 27, there is no evidence to suggest that the captured vulnerabilities can be reproduced and exploited on API levels 23 thru 27. Therefore, evaluations based on DroidBench should be aware of this limitation. *A prudent tool evaluation strategy is to equally consider both DroidBench and Ghera.*

3.5.2 DroidBench vs ICC-Bench

Observation 5 ICC-Bench uses only 3 APIs that are not used by DroidBench. *Since DroidBench uses almost all the APIs used by IccBench, DroidBench should be preferred over ICC-Bench.*

Observtion 6 ICC-Bench benchmarks were designed to run on API level 25 whereas DroidBench benchmarks were designed to run on API levels 22 and less. Consequently, two of the three APIs related to runtime checking of permissions are used by ICC-Bench but not by DroidBench as these APIs were introduced in the API level 23. *Therefore, if more current aspects of Android need to be considered, then IccBench should be used in conjunction with Ghera.*

3.5.3 DroidBench vs UBCBench

Observation 7 As per Table 3.6, UBCBench and DroidBench share 89 APIs. *Since only 10 APIs are unique to UBCBench and DroidBench uses almost all the APIs in UBCBench, DroidBench should be preferred over UBCBench.*

Observation 8 9 of the 10 APIs unique to UBCBench are related to general Java features and one API is related to SharedPreferences, a storage related feature in Android apps. This API is used by 90% of the real-world apps and discussed by close to 1000 Android security related Stack Overflow posts which makes this API highly relevant. So, *UBCBench should be considered in conjunction with DroidBench for tools that target vulnerabilities stemming from the use of SharedPreferences.*

3.5.4 Ghera vs ICC-Bench

Observation 9 Ghera covers most of the ICC related APIs used by ICC-Bench as seen by the fact that Ghera uses 42 of the 70 APIs used by ICC-Bench. Further, Ghera uses 476 APIs not used by ICC-Bench. *Since Ghera uses more than 50% of the APIs in ICC-Bench and is not limited to ICC, Ghera should be preferred over ICC-Bench.*

Observation 10 ICC-Bench uses 28 APIs not used by Ghera. All 28 are relevant and 25 of these APIs are used by more than 60% of real-world apps. Moreover, 23 of these APIs are related to ICC which is to be expected since ICC-Bench focuses on ICC. *Since ICC-Bench uses a non-trivial number of ICC-related relevant APIs not used by Ghera, ICC-Bench should be considered in conjunction with Ghera, especially if the the tool being evaluated is ICC focussed.*

3.5.5 Ghera vs UBCBench

Observation 11 As per Table 3.6, UBCBench and Ghera share 85 APIs and UBCBench has only 14 unique APIs. *Since Ghera uses almost all the APIs used by UBCBench, Ghera should be preferred over UBCBench. .*

Observation 12 6 of the 14 APIs unique to UBCBench are related to SharedPreferences and ICC while the remaining 8 are related to general Java features. *Since all 14 APIs are used by more than 2000 real-world apps and 10 of the 14 APIs are used by 60% of the real-world apps or more, UBCBench should be considered in conjunction with Ghera when evaluating tools that target vulnerabilities stemming from the use of SharedPreferences.*

Observation 13 Similar to DroidBench, the benchmarks in UBCBench were designed to run on API Level 19. Therefore, the authenticity of these benchmarks on API levels 23 thru 27 is unknown. Consequently, since DroidBench uses almost all of the APIs used by UBCBench, *the combination of Ghera and DroidBench should be preferred over the combination of Ghera and UBCBench.*

3.6 Threats to Validity

API usage used as a metric to measure representativeness of the considered benchmarks is a weak measure of manifestation of a vulnerability. It helps establish a correlation between the manifestation of the vulnerabilities in the benchmarks and the real-world apps. However, it does not prove that the vulnerability captured in a benchmark definitively exists in a large number of real-world apps. A stronger relationship can be established by using more complex and hard to measure aspects such as API usage context, security requirements of data, and data/control flow path between API uses. The influence of these aspects can be verified by using them to measure representativeness.

While a large sample of real-world apps has been used in this evaluation, there is a skew in the distribution of the apps across API levels (i.e., recent API levels tend to have lesser apps). Hence, API level might affect the results and observations. However, this is unlikely since APIs have not drastically changed since API level 23.

The baseline app from which the APIs to be ignored for representativeness calculation was determined can introduce bias based on how the baseline app was created. The effect of this bias can be measured and mitigated by using a different baseline app.

I used Stack Overflow discussions to identify relevant and security-related APIs used by a benchmark suite. Specifically, I used the occurrence of APIs in posts to associate posts to APIs. Since an occurrence of an API does not always imply the discussion of the API, the reported numbers of posts discussing an API may be inflated. This issue can be mitigated by using richer search techniques to associate posts to APIs to mitigate this issue. Similarly, the identification of tags associated with posts, to identify Android security-related posts, can be inaccurate due to incorrect tagging of posts. This threat can be addressed by using information retrieval techniques to identify incorrect tagging.

3.7 Evaluation Artifacts

I used the version/bundle of DroidBench and IccBench benchmarks available under *DroidBench (extended)* (MD5Sum 9a165494eec309ff49f1b72895308a13) and *ICC-Bench 2.0* (MD5Sum: d479d07c94a9415868b420c1f289a0b2) sections at https://github.com/FoelliX/ReproDroid.
The version of UBCBench I used is available at https://github.com/LinaQiu/UBCBench.
The version of Ghera used is available at
https://bitbucket.org/secure-it-i/android-app-vulnerability-benchmarks/src/Jan2019/.

The raw and processed along with scripts used in this study are available at https://bitbucket.org/account/user/secure-it-i/projects/BENCHPRESS.

3.8 Related Work

While there has been considerable interest in developing solutions for detecting vulnerabilities in Android apps, very few efforts are focused on developing and measuring benchmarks used to evaluate the effectiveness of the solutions. Only recently, Pauck et al. [52] developed a framework for verifying the authenticity of benchmarks in DroidBench and IccBench and refining them. In a similar vein, Qiu et al. [48] discovered that a few benchmarks in DroidBench and IccBench captured multiple aspects, which made their evaluation of the effectiveness of static taint analysis tools difficult. Therefore, they developed UBCBench and used it in conjunction with DroidBench and IccBench in their evaluation. In contrast, this effort focuses on measuring the representativeness of benchmark suites along with comparing them and identifying gaps in them.

Other efforts have used API usage to categorize vulnerabilities affecting Android apps. For example, Gorla et al. [59] used an app's description from app markets to infer its *advertised* behavior and the APIs used by the app to determine any anomalies. Similarly, Sadeghi et al. [60] measured the likelihood of a vulnerability pattern occurring in an Android app based on the app's source code patterns and API usage patterns. Distinct from such evaluations, the goal of our effort is to study benchmark suites and not Android apps or solutions related to Android app security.

Stack Overflow has been used in the past as a source for understanding security issues in Android apps. Stevens et al. [61] used Stack Overflow to study the relationship between the popularity of a *permission* and the number of times a permission is overused in an app. Similarly, Vasquez et al. [62] used Stack Overflow posts related to mobile development to understand the issues discussed by mobile app developers. In a similar vein, this effort also used Stack Overflow to identify security-related APIs. However, Stack Overflow data was used as an enabler and was not the focus of the evaluation.

3.9 Conclusion

In this chapter, I examined how well do existing Android app vulnerability benchmark suites represent real world apps in terms of the manifestation of vulnerabilities. I considered DroidBench, Ghera, ICC-Bench, and UBCBench benchmark suites and used API usage as a metric to measure representativeness.

I discovered that these benchmark suites are representative of real-world apps. Based on considered metrics, DroidBench was the most representative benchmark suite followed by Ghera, UBCBench, and ICC-Bench. Finally, in the context of tool evaluations, the results suggest that DroidBench and Ghera should be considered equally but ICC-Bench and UBCBench could be used to complement/strengthen the evaluations that use DroidBench and Ghera.

Chapter 4

Analyzing Android App Vulnerability Detection Tools

The security of mobile apps is crucial to the safety of its users, considering the vast amount of personal information these apps have access to and the critical tasks they perform. Hence, there is a growing need to ensure that apps are hardened against attacks from malicious apps. To address this need, in the last decade, researchers have developed a plethora of tools and techniques to enable app developers to detect vulnerabilities in Android apps [18, 21]. These solutions are based on:

Static Analysis is a technique where an app's compiled or source code is translated to an intermediate representation (IR) conducive for analysis. The IR is further analyzed using techniques such as control-flow, data-flow, and points-to analysis to detect vulnerabilities.

Dynamic Analysis is a technique to monitor an app's execution in a controlled environment and collect execution traces. These traces are analyzed further to identify vulnerabilities.

Despite the focus on developing tools and techniques to secure Android apps, there has been no comprehensive effort to evaluate them. The lack of such evaluations creates a gap between research and practice. The fact that many known vulnerabilities still occur in real-world apps [7, 8, 10] suggests that either existing tools and techniques are not effective in detecting such vulnerabilities, or they are not being used by developers to secure/harden their apps. Hence, to understand the state-of-the-art in Android app security analysis, in this chapter, I review 64 tools and evaluate the effectiveness of 14 vulnerability detection tools in detecting known Android app vulnerabilities. The results and observations from this evaluation can be used by tool developers to identify gaps in their tools, app developers to select the appropriate tools to secure their apps, and to guide future research related to Android app security.

4.1 Motivation

Android apps are often developed by small teams with limited resources [5, 6]. Hence, they cannot focus on all aspects of app development equally. Therefore, there is a need to develop automatic techniques to help detect and fix vulnerabilities in apps. Further, Android app developers are still unaware about security issues and how they transpire in Android apps [6]. Hence, solutions related to securing Android apps need to be usable off-the-shelf with no or minimal configuration.

In this context, numerous tools and techniques have been proposed to help app developers secure their apps. Consequently, in recent years, efforts have been made to assess the effectiveness of the proposed solutions [18, 49, 52, 63]. However, these evaluations are subject to the one or more of the following limitations:

1. Study tools in the reported literature without executing or using them
2. Exercise a small number of tools
3. Consider only academic tools
4. Consider tools that use a specific underlying techniques (e.g., static taint analysis)
5. Use technique specific benchmarks

6. Use benchmarks whose representativeness has not been established

7. Use random real-world apps that have no evidence of presence/absence of vulnerabilities.

Individual tool evaluations also suffer from the aforementioned limitations. Additionally, individual tool evaluations are focused on proving the effectiveness of their tools in detecting specific vulnerabilities [19, 20, 46]. While such evaluations are necessary from the perspective of the tools, it does not help understand the effectiveness of existing tools in detecting previously known vulnerabilities. Hence, the results from these evaluations does not help app developers choose the tools appropriate to secure their apps.

Considering the limitations of existing evaluation approaches, there is a need to systematically evaluate the effectiveness of Android app vulnerability detection tools in detecting known vulnerabilities. Hence, I experimented to evaluate the effectiveness of vulnerability detection tools for Android apps. In this chapter, I present the details and results of the experiment.

4.2 Evaluation Strategy

The first step was to collect the tools. I collected the tools from a variety of sources such as research papers [18, 21, 63], repositories [64], and blog posts [65] that collated information about Android security analysis solutions. From these sources, I considered 64 solutions and classified them as follows:

1. *Tools vs Frameworks: Tools* detect a fixed set of security issues. While they can be applied immediately, they are limited to detecting a fixed set of issues. On the other hand, *frameworks* facilitate the creation of tools that can detect specific security issues. While they are not immediately applicable to detect vulnerabilities and involve effort to create tools, they enable detection of a relatively open set of issues.

2. *Free vs Commercial:* Solutions are available either freely or for a fee.

3. *Maintained vs Unmaintained:* Solutions are either actively maintained or unmaintained (i.e., few years of development dormancy). Typically, unmaintained solutions do not support currently supported versions of Android. This is also true of a few maintained solutions.

4. *Vulnerability Detection vs Malicious Behavior Detection:* Solutions either detect vulnerabilities in an app or flag signs of malicious behavior in an app. The former is typically used by app developers while the latter is used by app stores and end users.

5. *Static Analysis vs Dynamic Analysis:* Solutions that rely on *static analysis* analyze either source code or Dex bytecode of an app and provide verdicts about possible security issues in the app. Since static analysis abstracts the execution environment, program semantics, and interactions with users and other apps, solutions reliant on static analysis can detect issues that occur in a variety of settings. However, since static analysis may incorrectly consider invalid settings due to too permissive abstractions, they are prone to high false positive rate.

   In contrast, solutions that rely on *dynamic analysis* execute apps and check for security issues at runtime. Consequently, they have a very low false positive rate. However, they are often prone to a high false negative rate because they may fail to explore specific settings required to trigger security issues in an app.

6. *Local vs Remote:* Solutions are available as executables or as sources from which executables can be built. These solutions are installed and executed locally by app developers.

   Solutions are also available remotely as web services (or via web portals) maintained by solution developers. App developers use these services by submitting the APKs of their apps for analysis and later accessing the analysis reports. Unlike local solutions, app developers are not privy to what happens to their apps when using remote solutions.

4.2.1 Benchmark Selection

Past efforts focused on the creation of benchmarks have considered certain criteria to ensure/justify the benchmarks are useful. For instance, database related benchmarks have considered relevance, scalability, portability, ease of use, ease of interpretation, functional coverage, and selectivity coverage [56]; web services related benchmarks have considered criteria such as repeatability, portability, representativeness, non-intrusiveness, and simplicity [66].

In a similar spirit, for evaluations of security analysis tools to be useful to tool users, tool developers, and researchers, evaluations should be based on vulnerabilities (consequently, benchmarks) that are *valid* (i.e., will result in a weakness in an app), *general* (i.e., do not depend on uncommon constraints such as rooted device or admin access), *exploitable* (i.e., can be used to inflict harm), and *current* (i.e., occur in existing apps and can occur in new apps).

The vulnerabilities captured in Ghera benchmarks have been previously reported in the literature or documented in Android documentation; hence, they are *valid*. These vulnerabilities can be verified by executing the benign and malicious apps in Ghera benchmarks on vanilla Android devices and emulators; hence, they are *general* and *exploitable*. These vulnerabilities are *current* as they are based on Android API levels that are currently supported by Android.

Due to these characteristics and the salient characteristics of Ghera — *tool and technique agnostic, authentic, feature specific, contextual (lean), version specific, duality, and representative* — described in Chapter 2 and Chapter 3, Ghera is well-suited for this evaluation.

At the time of this evaluation Ghera had 42 benchmarks and each benchmark contained apps with minimum API level 19 and target API level 25. Since then, Ghera has grown to 60 benchmarks and each benchmark contains apps with minimum API level 23 and target API level 27. The API level mismatch should not be a problem since API level 27 was released in 2019 and most vulnerability detection tools were designed for API levels 25 and less.

4.2.2 Tool Selection

To select tools for this evaluation, I first screened the considered 64 solutions by reading their documentation and any available resources. I rejected 5 solutions because they were not well documented (e.g., no documentation, lack of instructions to build and use tools). This was necessary to eliminate human bias resulting from the effort involved in discovering how to build and use a solution (e.g., DroidLegacy [67], BlueSeal [68]). I rejected AppGuard [69] because its documentation was not in English. I rejected 6 solutions such as Aquifer [70], Aurasium [71], and FlaskDroid [72] intended to enforce security policy. Such solutions enforce security policies at the platform level. Hence, they are not geared to detect vulnerabilities in apps that run on vanilla Android. Since the focus of this study was to evaluate the effectiveness of tools in detecting vulnerabilities known to occur in apps running on vanilla Android, I rejected such solutions.

Of the remaining 52 solutions, I selected solutions based on the first four classifications mentioned previously.

In terms of tools vs frameworks, I focused on solutions that could readily detect vulnerabilities with minimal adaptation (i.e., use it off-the-shelf) instead of having to build an extension to detect a specific vulnerability. The rationale was to eliminate human bias and errors involved in identifying, creating, and using the appropriate adaptations. Further, I wanted to mimic a *simple developer workflow*: procure/build the tool based on APIs it tackles and the APIs used in an app, follow its documentation, and apply it to the app. Consequently, I rejected 16 tools that only enabled security analysis (e.g., Drozer [73], ConDroid [74]). When a framework provided pre-packaged extensions to detect vulnerabilities, I selected such frameworks and treated each such extension as a distinct tool. For example, I selected Amandroid [20] framework as it comes with seven pre-packaged vulnerability detection extensions (i.e., data leakage, intent injection, comm leakage, password tracking, OAuth tracking, SSL misuse, and crypto misuse) that can be used as off-the-shelf tools.

In terms of free vs commercial, I rejected AppRay [75] as it was a commercial solution. While AppCritique [76] was a commercial solution, a feature-limited version of it was available for free. So, I decided to evaluate the free version and did not reject AppCritique.

In terms of maintained vs unmaintained, I focused on selecting only maintained tools. So, I rejected AndroWarn [77] and ScanDroid [78] tools as they were not updated after 2013. In a similar vein, since I was focused on currently supported Android API levels, I rejected TaintDroid [22] as it supported only API levels 18 or below.

In terms of vulnerability detection and malicious behavior detection, I selected only vulnerability detection tools and rejected 6 malicious behavior detection tools.

Next, I focused on tools that could be applied as is without extensive configuration (or inputs). The rationale was to eliminate human bias and errors involved in identifying and using appropriate configurations. So, I rejected tools that required additional inputs to detect vulnerabilities. Specifically, I rejected Sparta [79] as it required analyzed apps to be annotated with security types.

Next, I focused on the applicability of tools to Ghera benchmarks. I considered only tools that claimed to detect vulnerabilities stemming from APIs covered by at least one category in Ghera benchmarks. For such tools, based on my knowledge of Ghera benchmarks and shallow exploration of the tools, I assessed if the tools were indeed applicable to the benchmarks in the covered categories. The shallow exploration included checking if the APIs used in Ghera benchmarks were mentioned in any lists of APIs bundled with tools (e.g., the list of information source and sink APIs bundled with HornDroid and FlowDroid). In this regard, I rejected 2 tools (and 1 extension): a) PScout [16] focused on vulnerabilities related to over/under use of permissions and the only permission related benchmark in Ghera was not related to over/under use of permissions and b) LetterBomb and Amandroid's OAuth tracking extension ($Amandroid_5$) as they were not applicable to any Ghera benchmark.[1]

Of the remaining 23 tools, for tools that could be executed locally, I downloaded the latest official release of the tools (e.g., Amandroid).

If an official release was not available, then I downloaded the most recent version of the tool (executable or source code) from the master branch of its repository (e.g., AndroBugs

[1]While Ghera did not have benchmarks that were applicable to some of the rejected tools at the time of this evaluation, it currently has such benchmarks that can be used in subsequent iterations of this evaluation.

[80]). I then followed the tool's documentation to build and set up the tool. If I encountered issues during this phase, then I tried to fix them; specifically, when issues were caused by dependency on older versions of other tools (e.g., HornDroid failed against real-world apps as it was using an older version of `apktool`, a decompiler for Android apps), incorrect documentation (e.g., documented path to the DIALDroid[81] executable was incorrect), and incomplete documentation (e.g., IccTA's documentation did not mention the versions of required dependencies). The fixes were limited to being able to execute the tools and not to affect the behavior of the tool. I stopped trying to fix an issue and rejected a tool if I could not figure out a fix by interpreting the error messages and by exploring existing publicly available bug reports. This resulted in rejecting DidFail [82].

Of the remaining tools, I tested 18 local tools using the benign apps from randomly selected Ghera benchmarks A.2.1, A.2.2, A.8.1, and A.8.9 and randomly selected apps Offer Up, Instagram, Microsoft Outlook, and My Fitness Pal's Calorie Counter from Google Play store. Each tool was executed with each of the above apps as input on a 16 core Linux machine with 64GB RAM and with 15 minute time out period. If a tool failed to execute successfully on all of these apps, then I rejected the tool. Specifically, I rejected IccTA and SMV Hunter [83] because they failed to process the test apps by throwing exceptions. I rejected CuckooDroid [84] and DroidSafe [85] because they ran out of time or memory while processing the test apps.

For 9 tools that were available only remotely, I tested them by submitting the above test apps for analysis. If a tool's web service was unavailable, failed to process all of the test apps, or did not provide feedback within 30–60 minutes, then I rejected it. Consequently, I rejected 4 remote tools (e.g., TraceDroid [86]).

Table 4.1 lists the fourteen tools selected for evaluation along with their canonical reference. For each tool, the table reports the version (or the commit id) selected for evaluation, dates of its initial publication and latest update, whether it uses static analysis (S) or dynamic analysis (D) or both (SD), whether it runs locally (L) or remotely (R), and the time spent to set up tools on a Linux machine.

Tool	Commit Id / Version	Updated [Published]	S/D	L/R	A/N	H/E	Set Up Time (sec)
Amandroid [20]	3.1.2	2017 [2014]	S	L	A	E	3600
AndroBugs [80]	7fd3a2c	2015 [2015]	S	L	N	H	600
AppCritique [76]	?	? [?]	?	R	N	?	
COVERT [14]	2.3	2015 [2015]	S	L	A	E	2700
DevKnox [87]	2.4.0	2017 [2016]	S	L	N	H	600
DIALDroid[81]	25daa37	2018 [2016]	S	L	A	E	3600
FixDroid [88]	1.2.1	2017 [2017]	S	L	A	H	600
FlowDroid [19]	2.5.1	2018 [2013]	S	L	A	E	9000
HornDroid [51]	aa92e46	2018 [2017]	S	L	A	E	600
JAADAS [89]	0.1	2017 [2017]	S	L	N	H/E	900
MalloDroid [46]	78f4e52	2013 [2012]	S	L	A	H	600
Marvin-SA[2][90]	6498add	2016 [2016]	S	L	N	H	600
MobSF [91]	b0efdc5	2018 [2015]	SD	L	N	H	1200
QARK [92]	1dd2fea	2017 [2015]	S	L	N	H	600

Table 4.1: *Evaluated vulnerability detection tools. "?" denotes unknown information.* S *and* D *denote use of static analysis and dynamic analysis, respectively.* L *and* R *denote the tool runs locally and remotely, respectively.* A *and* N *denote academic tool and non-academic tool, respectively.* H *and* E *denote the tool uses shallow analysis and deep analysis, respectively. Empty cell denotes non-applicable cases.*

4.3 Experiment

Every selected vulnerability detection tool (including pre-packaged extensions treated as tools) was applied in its default configuration to the *benign* app and the *secure* app separately of every applicable Ghera benchmark (given in column 9 in Table 4.4). The influence of API level on the performance of tools was controlled by using APKs that were built with minimum API level of 19 and target API level of 25 (i.e., these APKs can be installed and executed with every API level from 19 thru 25).

The tools were executed on a 16 core Linux machine with 64GB RAM and with 15 minutes time out. For each execution, I recorded the execution time and any output reports, error traces, and stack traces. I then examined the output to determine the verdict and its veracity pertaining to the vulnerability.

4.3.1 Ensuring Fairness

Consider only supported versions of Android: FixDroid was not evaluated on secure apps in Ghera because Android Studio version 3.0.1 was required to build the secure apps in Ghera and FixDroid was available as a plugin only to Android Studio version 2.3.3. Further, since benchmarks A.1.3, A.2.13, A.2.14, and A.6.4 were added after Ghera was migrated to Android Studio version 3.0.1, FixDroid was not evaluated on these benchmarks; hence, FixDroid was evaluated on only 38 Ghera benchmarks.

Provide inputs as required by tools: COVERT and DIALDroiddetect vulnerabilities stemming from inter-app communications (e.g., collusion, compositional vulnerabilities). So, each of these tools was applied in its default configuration to 33 Ghera benchmarks that included malicious apps. For each benchmark, the malicious app was also provided as input together with the benign app and the secure app.

Consider multiple operational modes: JAADAS operates in two modes: fast mode in which only intra-procedural analysis is performed and full mode in which both intra- and inter-procedural analyses are performed. Since the modes can be selected easily, JAADAS was evaluated in both modes.

QARK can analyze the source code and the APK of an app. It decompiles the DEX bytecodes in an APK into source form. Since the structure of reverse engineered source code may differ from the original source code, which could affect the accuracy of QARK's verdicts, both APKs and source code were used to evaluate QARK.

Consider only supported API levels: Since the inception and evolution of tools are independent of the evolution of Android API levels, a tool may not support an API level (e.g., the API level is released after the tool was last developed/updated) and, hence, it may fail to detect vulnerabilities stemming from APIs introduced in such unsupported API levels. To control for this effect, I identified the API levels supported by tools.

Of the 14 tools, only 3 tools provide some information about the supported API levels

	Cumulative # of Considered Vulnerabilities									
Year	Crypto	ICC	Net	Perm	Store	Sys	Web	Total	# Tools	API Levels
2011	0	7	0	0	2	0	0	9	0	
2012	0	7	0	0	2	0	4	13	0	
2013	4	7	0	0	2	0	7	20	1	19
2014	4	11	0	1	6	4	8	34	1	21
2015	4	14	1	1	6	4	9	39	3	22, 23
2016	4	14	0	1	6	4	9	39	4	24, 25
2017	4	14	2	1	6	4	9	40	9	
2018	0	16	2	1	6	4	9	42	13[3]	

Table 4.2: *Cumulative number of considered vulnerabilities discovered until a specific year (inclusive).* Tools *column is cumulative number of evaluated tools published until a specific year (inclusive).* API Levels *column lists the API levels released in a specific year.*

(Android platform versions). Specifically, JAADAS documentation states that JAADAS will work for all Android platforms (as long as the platform version is provided as input). Amandroid was successfully used to process ICC-Bench benchmarks that target API level 25 [20]; hence, I inferred that Amandroid supports API level 25 and below. The DIALDroidtool repository contains versions of Android platform corresponding to API levels 3 thru 25 that are to be provided as input to the tool; hence, I inferred DIALDroidsupports API levels 19 thru 25.

In the absence of such information, I conservatively assumed the tools supported API levels 19 thru 25 and this assumption is fair because 1) API level 19 and 25 were released in October 2013 and December 2016, respectively, (see *API Levels* column in Table 4.2), 2) all tools were last updated in 2013 or later (see *Updated* column in Table 4.1), and 3) every Ghera benchmark APK used in the evaluation were built to run on every API level 19 thru 25.

Consider only applicable categories of vulnerabilities: While each tool could be evaluated against all of 42 considered known vulnerabilities, such an evaluation would be unfair as all tools are not geared to detect all kinds of vulnerabilities (e.g., cryptography related vulnerability vs storage related vulnerability). Further, during app development, tools are often selected based on their capabilities pertaining to platform features/APIs

	API (Vulnerability) Categories						
Tool	Crypto	ICC	Net	Perm	Store	Sys	Web
Amandroid	✓	✓	✓		✓		✓
AndroBugs *	✓	✓	✓	✓	✓	✓	✓
AppCritique *	✓	✓	✓	✓	✓	✓	✓
COVERT	✓	✓	✓	✓	✓	✓	✓
DevKnox *	✓	✓	✓	✓	✓	✓	✓
DIALDroid	✓	✓	✓	✓	✓	✓	✓
FixDroid	✓	✓	✓	✓	✓	✓	✓
FlowDroid		✓	✓		✓		
HornDroid	✓	✓	✓		✓		✓
JAADAS *	✓	✓	✓	✓	✓	✓	✓
MalloDroid							✓
Marvin-SA *	✓	✓	✓	✓	✓	✓	✓
MobSF *	✓	✓	✓	✓	✓	✓	✓
QARK *	✓	✓	✓	✓	✓	✓	✓

Table 4.3: *Applicability of vulnerability detection tools to various benchmark categories in Ghera. "✓" denotes the tool is applicable to the vulnerability category in Ghera. Empty cell denotes non-applicable cases. "*" identifies non-academic tools.*

being used in the app.

To control for this effect, for each tool, I identified the categories of vulnerabilities that it was applicable to and evaluated it against only the vulnerabilities from these categories. Even within categories, I ignored vulnerabilities if a tool was definitively inapplicable to them. For example, MalloDroid tool was evaluated only against SSL/TLS related vulnerabilities from Web category as the tool focuses on detecting SSL/TLS related vulnerabilities; see entry in *Web* column for MalloDroid in Table 4.4. For each tool, Table 4.3 reports the applicable Ghera benchmark categories.

Consider the existence of vulnerabilities: Expecting a tool to detect vulnerabilities that did not exist when the tool was developed/updated would be unfair. In terms of the purpose of this evaluation, this is not a concern as the evaluation is less focused on individual tools and more focused on assessing the effectiveness of the existing set of vulnerability detection tools against considered known vulnerabilities. In terms of the execution of this evaluation, this is not a concern as almost all of the considered vulnerabilities (39 out of 42) were discovered before 2016 (see *Total* column in Table 4.2) and almost all of the evaluated tools (at least 10 out of 14) were updated in or after 2016 (see *# Tools* column in Table 4.2).

4.4 Observation and Open Questions

In this section, I make interesting observations about the current state of tools. Further, I raise open questions based on the data from the experiment, to help guide future research.

4.4.1 Tools Selection

Open Questions 1 & 2 Of the considered 64 solutions, 17 tools (including Amandroid) were intended to enable security analysis of Android apps. This is a bit over 25% of security analysis tools considered in this evaluation. Further, I have found these tools be useful in my research workflow (e.g., Drozer, MobSF). Hence, studying these tools may be useful. Specifically, exploring two mutually related questions: *1) how expressive, effective, and easy-to-use are tools that enable security analysis?* and *2) are Android app developers and security analysts willing to invest effort in such tools?* may help both tool users and tool developers.

Observation 1 I rejected 39% of tools (9 out of 23) considered in deep selection. Even considering the number of instances where the evaluated tools failed to process certain benchmarks (see numbers in square brackets in Table 4.4), such a low rejection rate is rather impressive. This suggests *tool developers are putting in effort to release robust security analysis tools.* This number can be further improved by distributing executables (where applicable), providing complete and accurate build instructions (e.g., versions of required dependencies) for local tools, providing complete and accurate information about execution environment (e.g., versions of target Android platforms), and publishing estimated turn around times for remote tools.

Observation 2 If the sample of tools included in this evaluation is representative of the population of Android app security analysis tools, then *almost every vulnerability detec-*

tion tool for Android apps relies on static analysis (i.e., 13 out of 14); see *S/D* column in Table 4.1.

Observation 3 Every vulnerability detection tool publicly discloses the category of vulnerabilities it tries to detect. Also, almost all vulnerability detection tools are available as locally executable tools (i.e., 13 out of 14; see *L/R* column in Table 4.1). So, *vulnerability detection tools are open with regards to their vulnerability detection capabilities.* The likely reason is to inform app developers how the security of apps improves by using a vulnerability detection tool and encourage the use of appropriate vulnerability detection tools.

Observation 4 Ignoring tools with unknown update dates ("?" in column 3 of Table 4.1) and considering the evaluation was conducted between June 2017 and May 2018, 9 out of 13 tools are less than 1.5 years old (2017 or later) and 12 out of 13 are less than or about 3 years old (2015 or later). Hence, the selected tools can be considered as current. Consequently, *the resulting observations are highly likely to be representative of the current state of the freely available Android app security analysis tools.*

4.4.2 Vulnerability Detection Tools

Table 4.4 summarizes the results from executing tools to evaluate their effectiveness in detecting different categories of vulnerabilities. In the table, the number of vulnerabilities (benchmarks) that existed when a tool was last developed/updated before this evaluation and the number of vulnerabilities that a tool was applied to in this evaluation is given by *Existed* and *Applied* columns, respectively.

Every Ghera benchmark is associated with exactly one unique vulnerability v, and its benign app exhibits v while its secure app does not exhibit v. So, for a tool, for each applicable benchmark, I classified the tool's verdict for the benign app as either true positive (i.e., v was detected in the benign app) or false negative (i.e., v was not detected in the benign app). I classified the tool's verdict for the secure app as either true negative (i.e., v was not detected in a secure app) or false positive (i.e., v was detected in a secure app). Columns *TP, FN*,

Tool	Crypto	ICC	Net	Perm	Store	Sys	Web	# Benchmarks		Benign		Secure	Other
	(4)	(16)	(2)	(1)	(6)	(4)	(9)	Applied	Existed	TP	FN	TN	
Amandroid$_1$		7/**0**/**9**/3	1/**0**/**1**/0		4/**0**/**1**/0 {1}		6/**0**/**3**/0	15	13	1	14	14	3
Amandroid$_2$		X			1/**0**/**2**/0 [3]		5/**0**/**0**/0 [4]	25	23	0	3	3	0
Amandroid$_3$		8/**0**/**8**/2	1/**0**/**1**/0		4/**0**/**2**/0		6/**0**/**3**/0	14	12	0	14	14	2
Amandroid$_4$		13/**0**/**3**/2						3	3	0	3	3	2
Amandroid$_6$							6/**0**/**3**/0	3	3	0	3	3	0
Amandroid$_7$	0/**2**/**2**/0							4	4	2	2	4	0
AndroBugs *	N	0/**2**/**14**/3	N	0/**1**/**0**/0	N	0/**4**/**0**/0	0/**4**/**5**/1	42	39	11	31	42	4
AppCritique *	0/**2**/**2**/0	N	N	N	0/**3**/**3**/0	N	0/**2**/**7**/0	42	?	7	35	42	0
COVERT	N	N	1/**0**/**1**/0	N	N	N	8/**0**/**1**/0	33	30	0	33	33	0
DevKnox *	0/**1**/**3**/0	N	N	N	N	D	N	42	40	5	37	38	0
DIALDroid	N	N	1/**0**/**1**/0	N	N	N	8/**0**/**1**/0	33	33	0	33	33	0
FixDroid	0/**1**/**2**/2	13/**1**/**0**/0	N	N	0/**1**/**4**/0	0/**4**/**0**/0	0/**2**/**7**/0	25	25	9	16	-	2
FlowDroid		N	N		N			24	24	0	24	24	0
HornDroid	N	0/**1**/**15**/7	N		0/**0**/**6**/1		0/**0**/**9**/1	37	37	1	36	37	9
JAADAS$_1$*	N	0/**2**/**14**/0	N	N	N	N	0/**4**/**5**/1	42	40	6	36	42	1
JAADAS$_2$*	N	0/**2**/**14**/0	N	N	N	N	0/**4**/**5**/1	42	40	6	36	42	1
MalloDroid			X				5/**0**/**1**/0 [3]	4	4	0	1	1	0
Marvin-SA *	0/**1**/**3**/0	0/**5**/**11**/3	N	0/**1**/**0**/0	0/**0**/**6**/2	0/**4**/**0**/0	0/**4**/**5**/0	42	39	15	27	42	5
MobSF *	0/**1**/**3**/0	0/**5**/**11**/0	N	0/**1**/**0**/1	0/**1**/**5**/0	0/**4**/**0**/0	0/**3**/**6**/0	42	42	15	27	42	1
QARK$_1$*	N	0/**3**/**13**/0	N	0/**1**/**0**/0	N	0/**4**/**0**/0	0/**2**/**7**/0	42	40	10	32	42	0
QARK$_2$*	N	0/**3**/**13**/0	N	0/**1**/**0**/0	N	0/**4**/**0**/0	0/**2**/**7**/0	42	40	10	32	42	0
# Undetected	1	5	2	0	2	0	2						

Table 4.4: *Results from evaluating vulnerability detection tools. The number of benchmarks in each category is mentioned in parentheses. In each category, empty cell denotes the tool is inapplicable to any of the benchmarks, N denotes the tool flagged both benign and secure apps as not vulnerable in every benchmark, X denotes the tool failed to process any of the benchmarks, and D denotes the tool flagged both benign and secure apps as vulnerable in every benchmark. H/I/J/K denotes the tool was inapplicable to H benchmarks, flagged benign app as vulnerable and secure app as not vulnerable in I benchmarks, flagged both benign and secure app as not vulnerable in J benchmarks, and reported non-existent vulnerabilities in benign or secure apps in K benchmarks. The number of benchmarks that a tool failed to process is mentioned in square brackets. The number of benchmarks in which both benign and secure apps were flagged as vulnerable is mentioned in curly braces. "-" denotes not applicable cases. "*" identifies non-academic tools.*

and *TN* in Table 4.4 report true positives, false negatives, and true negatives, respectively. False positives are not reported in the table as none of the tools except DevKnox (observe the D for DevKnox under *System* benchmarks in Table 4.4) and *data leakage* extension of Amandroid (observe the {1} for Amandroid$_1$ under *Storage* benchmarks in Table 4.4) provided false positive verdicts. Reported verdicts do not include cases in which a tool failed to process apps.

Observation 5 *Most of the tools (10 out of 14) were applicable to every Ghera benchmark*; see # *Applicable Benchmarks* column in Table 4.4. With the exception of MalloDroid, the rest of the tools were applicable to 24 or more Ghera benchmarks. This observation is also true of Amandroid if the results of its pre-packaged extensions are considered together.

Observation 6 Based on the classification of the verdicts, 4 out of 14 tools detected none of the vulnerabilities captured in Ghera ("0" in *TP* column in Table 4.4) considering all extensions of Amandroid as one tool. Even in case of tools that detected some of the vulnerabilities captured in Ghera, none of the tools individually detected more than 15 out of the 42 vulnerabilities; see the numbers in *TP* column and the number of N's under various categories in Table 4.4. This suggests, *in isolation, current tools are very limited in their ability to detect known vulnerabilities captured in Ghera.*

Observation 7 For 11 out of 14 tools,[4] the number of false negatives was greater than 70% of the number of true negatives; see *FN* and *TN* columns in Table 4.4.[5] This proximity between the number of false negatives and the number of true negatives suggests two possibilities: *most tools prefer to report only valid vulnerabilities (i.e., be conservative)* and *most tools can only detect specific manifestations of vulnerabilities.* Both these possibilities limit the effectiveness of tools in assisting developers to build secure apps.

[4]AndroBugs, Marvin-SA, and MobSF were the exceptions.

[5]I considered all variations of a tool as one tool (e.g., JAADAS). FixDroid was not counted as it was not evaluated on secure apps in Ghera.

Observation 8 Tools make claims about their ability to detect specific vulnerabilities or class of vulnerabilities. So, I examined such claims. For example, while both COVERT and DIALDroidclaimed to detect vulnerabilities related to communicating apps, neither detected such vulnerabilities in any of the 33 Ghera benchmarks that contained a benign app and a malicious app. Also, while MalloDroid focuses solely on SSL/TLS related vulnerabilities, it did not detect any of the SSL vulnerabilities present in Ghera benchmarks. I observed similar failures with FixDroid. (See numbers in *# Applicable Benchmarks* and *TP* columns for COVERT, DIALDroid, FixDroid, and MalloDroid in Table 4.4.) This suggests *there is a gap between the claimed capabilities and the observed capabilities of tools that could lead to vulnerabilities in apps.*

Observation 9 Different tools use different kinds of analysis under the hood to perform security analysis. Tools such as QARK, Marvin-SA, and AndroBugs rely on *shallow analysis* (e.g., searching for code smells/patterns) while tools such as Amandroid, FlowDroid, and HornDroid rely on *deep analysis* (e.g., data flow analysis); see *H/E* column in Table 4.1. By considering this aspect with the verdicts provided by the tools (see *TP* and *FN* columns in Table 4.4), I observe *tools that rely on deep analysis report fewer true positives and more false negatives than tools that rely on shallow analysis.*

Interestingly, among the evaluated tools, most academic tools relied on deep analysis while most non-academic tools relied on shallow analysis; see *H/E* columns in Table 4.1 and tools marked with * in Table 4.4.

Open Questions 3 & 4 A possible reason for the poor performance of deep analysis tools could be they often depend on extra information about the analyzed app (e.g., a custom list of sources and sinks to be used in data flow analysis) and I did not provide such extra information in this evaluation. However, JAADAS was equally effective in both fast (intra-procedural analysis) and full (intra- and inter-procedural analyses) modes, that is, true positives, false negatives, and true negatives remained unchanged across modes. Also, FixDroid was more effective than other deep analysis tools even while operating within

Year	Crypto	ICC	Net	Store	Web	Total
2011	-	2	-	-	-	2
2012	-	-	-	-	-	-
2013	1	-	-	-	1	2
2014	-	1	-	2	1	4
2015	-	-	1	-	-	1
2016	-	-	-	-	-	-
2017	-	-	1	-	-	1
2018	-	2	-	-	-	2

Table 4.5: *Number of undetected vulnerabilities discovered in a specific year. "-" denotes zero undetected vulnerabilities were discovered.*

an IDE; it was the fifth best tool in terms of the number of true positives. Clearly, in this evaluation, shallow analysis tools seem to out perform deep analysis tools. This raises two related questions: *3) are Android app security analysis tools that rely on deep analysis effective in detecting vulnerabilities in general?* and *4) are the deep analysis techniques used in these tools well suited in general to detect vulnerabilities in Android apps?* These questions are pertinent because Ghera benchmarks capture known vulnerabilities and the benchmarks are small/lean in complexity, features, and size (i.e., less than 1000 lines of developer created Java and XML files) and yet deep analysis tools failed to detect the vulnerabilities in these benchmarks.

Observation 10 From the perspective of vulnerabilities, every vulnerability captured by *Permission* and *System* benchmarks were detected by some tool. However, none of the 2 vulnerabilities captured by *Networking* benchmarks were detected by any tool. Considering all vulnerabilities, 12 of 42 known vulnerabilities captured in Ghera were not detected by any tool (false negatives); see # *Undetected* row in Table 4.4. In other words, *using all tools together is not sufficient to detect the known vulnerabilities captured in Ghera.*

In line with the observation made in Section 4.3.1 based on Table 4.2 – most of the vulnerabilities captured in Ghera were discovered before 2016, most of the vulnerabilities (9 out of 12) not detected by any of the evaluated tools were discovered before 2016; see Table 4.5.

Open Questions 5 & 6 Of the 42 vulnerabilities, 30 vulnerabilities were detected by 14 tools with no or minimal configuration. This is collectively impressive. To build on this, two questions are worth exploring: *5) with reasonable configuration effort, can the evaluated tools be configured to detect the undetected vulnerabilities?* and *6) would the situation improve if vulnerability detection tools rejected during tools selection are also used to detect vulnerabilities?*

Observation 11 Of the 14 tools, 8 tools reported vulnerabilities that were not the focus of Ghera benchmarks; see *Other* column in Table 4.4. Upon manual examination of these benchmarks, I found none of the reported vulnerabilities in the benchmarks. Hence, *with regards to vulnerabilities not captured in Ghera benchmarks, tools exhibit a high false positive rate.*

Observation 12 To understand the above observations, I identified the relevant APIs and security-related APIs used by the version of Ghera used in this evaluation. *Relevant APIs* and *security-related APIs* mean the same as *Considered APIs* and *security-related* APIs defined in Section 3.3. However, in this evaluation, security-related APIs were identified manually and not from Stack Overflow posts as described in Section 3.3. I compared the sets of relevant APIs used in benign apps (601 APIs) and secure apps (602 APIs). I found that 587 APIs were common to both sets, while 14 and 15 APIs were unique to benign apps and secure apps. When I compared security-related APIs used in benign apps (117 APIs) and secure apps (108 APIs), 108 were common to both sets and 9 were unique to benign apps. This suggests the benign apps (real positives) and secure apps (real negatives) are similar in terms of the APIs they use and different in terms of how they use APIs (i.e., different arguments/flags, control flow context). This implies *tools should consider aspects beyond the presence of APIs to successfully identify the presence of vulnerabilities captured in Ghera.*

Observation 13 I partitioned the set of 601 relevant APIs and the set of 117 security-related APIs used in benign apps into three sets: 1) *common APIs* that appear in both true positive benign apps (flagged as vulnerable) and false negative benign apps (flagged as secure), 2) *TP-only APIs* that appear only in true positive benign apps, and 3) *FN-only APIs* that appear only in false negative benign apps. The sizes of these sets in order in each partition were 440, 108, and 53, and 60, 39, and 18, respectively. For both relevant and security-related APIs, the ratio of the number of TP-only APIs and the number of FN-only APIs is similar to the ratio of the number of true positives and the number of false negatives (i.e., $108/53 \approx 39/18 \approx 30/12$). This relation suggests, *to be more effective in detecting vulnerabilities captured in Ghera, tools should be extended to consider FN-only APIs.* Since vulnerabilities will likely depend on the combination of FN-only APIs and common APIs, such extensions should also consider common APIs.

Observation 14 I compared the sets of relevant APIs used in the 12 false negative benign apps (flagged as secure) and all of the secure apps (real negatives). While 491 APIs were common to both sets, only 2 APIs were unique to benign apps. In the case of security-related APIs, 77 APIs were common while only 1 API was unique to secure apps. This suggests, in terms of APIs, false negative benign apps are very similar to secure apps. Consequently, *tools need to be more discerning to correctly identify benign apps in Ghera as vulnerable (i.e., reduce the number of false negatives) without incorrectly identifying secure apps as vulnerable (i.e., increase the number of false positives).*

Observation 15 I measured Ghera's representativeness in terms of APIs that were used in true positive benign apps (i.e., common APIs plus TP-only APIs) and the APIs that were used in false negative benign apps. I considered TP-only (FN-only) APIs along with common APIs as vulnerabilities may depend on the combination of TP-only (FN-only) APIs and common APIs. I found that at least 83% (457 out of 548) of relevant APIs and 70% (70 out of 99) of security-related APIs used in true positive benign apps were used in at least 50% of apps in the real-world app sample.[6] These numbers are 84% (416 out of 493) and 61% (48 out of 78) in case of false negative benign apps. This result suggests that *the effectiveness of tools in detecting vulnerabilities captured by Ghera benchmarks will extend to real-world apps.*

Observation 16 In terms of the format of the app analyzed by tools, *all tools supported APK analysis.* A possible explanation for this is analyzing APKs helps tools cater to a wider audience: APK developers and APK users (i.e., app stores and end users).

Observation 17 For tools that completed the analysis of apps (either normally or exceptionally), the median run time was 5 seconds with the lowest and highest run times being 2 and 63 seconds, respectively. So, *in terms of performance, tools that complete analysis exhibit good run times.*

Observations Based on Evaluation Measures

Besides drawing observations from raw numbers, I also make observations based on evaluation measures.

While precision and recall are commonly used evaluation measures, they are biased — "they ignore performance in correctly handling negative cases, they propagate the underlying marginal prevalences (real labels) and biases (predicted labels), and they fail to take account the chance level performance". So, I used informedness and markedness, which are unbiased variants of recall and precision, respectively, as evaluation measures.

As defined by Powers, *Informedness* quantifies how informed a predictor is for the specified condition, and specifies the probability that a prediction is informed in relation to the condition (versus chance) [93]. *Markedness* quantifies how marked a condition is for the specified predictor and specifies the probability that a condition is marked by the predictor (versus chance). Quantitatively, informedness and markedness are defined as the difference

[6]This sample is different from the sample of real-world apps discussed in Chapter 3. Here, I collected a sample of 111K real-world apps with target API levels 19 thru 25 to match the API levels supported by the version of Ghera used in this evaluation.

Tool	Precision	Recall	Informedness	Markedness
Amandroid$_1$	0.500	0.067	0.000	0.000
Amandroid$_2$	–	0.000	0.000	–
Amandroid$_3$	–	0.000	0.000	–
Amandroid$_4$	–	0.000	0.000	–
Amandroid$_6$	–	0.000	0.000	–
Amandroid$_7$	1.000	1.000	1.000	1.000
AndroBugs *	1.000	0.262	0.262	0.575
AppCritique *	1.000	0.167	0.167	0.545
COVERT	–	0.000	0.000	–
DIALDroid	–	0.000	0.000	–
DevKnox *	0.556	0.119	0.024	0.062
FixDroid	1.000	0.231	–	0.000
FlowDroid	–	0.000	0.000	–
HornDroid	1.000	0.027	0.027	0.507
JAADAS *	1.000	0.143	0.143	0.538
MalloDroid	–	0.000	0.000	–
Marvin-SA *	0.938	0.357	0.333	0.540
MobSF *	1.000	0.310	0.310	0.592
QARK *	1.000	0.333	0.333	0.600

Table 4.6: *Precision, Recall, Informedness, and Markedness scores from tools evaluation. "–" denotes measure was undefined. "*" identifies non-academic tools.*

between true positive rate and false positive rate (i.e., $TP/(TP+FN)-FP/(FP+TN)$) and the difference between true positive accuracy and false negative accuracy, (i.e., $TP/(TP+FP)-FN/(FN+TN)$), respectively.[7] When they are positive, the predictions are better than chance (random) predictions. When these measures are zero, the predictions are no better than chance predictions. When they are negative, the predictions are perverse and, hence, worse than chance predictions.

In this evaluation, informedness is interpreted as *a measure of how informed (knowledgeable) is a tool about the presence and absence of vulnerabilities* (i.e., will a vulnerable/secure app be detected as vulnerable/secure?) and *markedness as a measure of the trustworthiness (truthfulness) of a tool's verdict about the presence and absence of vulnerabilities* (i.e., is an app that is flagged (marked) as vulnerable/secure indeed vulnerable/secure?)

[7]TP, FP, FN, and TN denote the number of true positive, false positive, false negative, and true negative verdicts, respectively.

Table 4.6 reports the informedness and markedness for the evaluated tools. It also reports precision and recall to help readers familiar with precision and recall but not with informedness and markedness. It reports the measures for each Amandroid plugin separately as each plugin was applied to different sets of benchmarks. It does not report measures for each variation of JAADAS and QARK separately as the variations for each tool were applied to the same set of benchmarks and provided identical verdicts. For tools that did not flag any app as vulnerable (positive), markedness was undefined. Informedness was undefined for FixDroid as it was not evaluated on secure (negative) apps.

Observation 18 Out of 13 tools, 6 tools were better informed than an uninformed tool about the presence and absence of vulnerabilities (i.e., $0.14 \leq$ informedness ≤ 0.33; see *Informedness* column in Table 4.6). These tools reported a relatively higher number of true positives and true negatives; see *TP* and *TN* columns in Table 4.4. At the extremes, while Amandroid$_7$ plugin was fully informed (i.e., informedness $= 1.0$), the remaining tools and Amandroid plugins were completely uninformed about the applicable vulnerabilities they were applicable to (i.e., informedness ≈ 0) as they did not report any true positives. This suggests *tools need to be much more informed about the presence and absence of vulnerabilities to be effective.*

Observation 19 Out of 14 tools, 8 tools provided verdicts that were more trustworthy than random verdicts (i.e., $0.5 \leq$ markedness; see *Markedness* column in Table 4.6). The verdicts from Amandroid$_7$ plugin could be fully trusted with regards to the applicable vulnerabilities (benchmarks) (i.e., markedness $= 1.0$). The verdicts of Amandroid$_1$ were untrustworthy (i.e., markedness $= 0$). The verdicts of FixDroid cannot be deemed untrustworthy based on markedness score because I did not evaluate FixDroid on secure apps. The remaining tools and Amandroid plugins did not flag any apps from any of the applicable benchmarks as vulnerable (i.e., no true positive verdicts). Unlike in the case of informedness, *while there are tools that can be trusted with caution, tools need to improve the trustworthiness of their verdicts to be effective.*

Both the uninformedness and unmarkedness (lack of truthfulness of verdicts) of tools could be inherent to techniques underlying the tools or stem from the use of minimal configuration when exercising the tools. So, while both possibilities should be explored, *the ability of tools to detect known vulnerabilities should be evaluated with extensive configuration before starting to improve the underlying techniques.*

Observation 20 In line with observation 9, *in terms of both informedness and markedness, shallow analysis tools fared better than deep analysis tools*; see *H/E* column in Table 4.1 and *Informedness* and *Markedness* columns in Table 4.6. Also, non-academic tools fared better than academic tools; see tools marked with * in Table 4.6. Similar to observation 9, these measures also reinforce the need to explore questions 3, 4, and 5.

4.5 Threats to Validity

Internal Validity While all the tools were executed using all possible options but with minimum or no extra configuration, I may not have used options or combinations of options that could result in more true positives and true negatives. The same is true of extra configuration required by certain tools (e.g., providing a custom list of sources and sink to FlowDroid).

Personal preferences for IDEs (e.g., Android Studio over Eclipse) and flavors of analysis (e.g., static analysis over dynamic analysis) could have biased how I addressed issues encountered while building and setting up tools. This could have affected the selection of tools and the reported set up times.

I have taken utmost care in using the tools, collecting their outputs, and analyzing their verdicts. However, bias along with commission errors (e.g., incorrect tool use, overly permissive assumptions about API levels supported by tools) and omission errors (e.g., missed data) could have affected the evaluation.

All of the above threats can be addressed by replicating this evaluation; specifically, by different experimenters with different biases and preferences and comparing their observations with the observations documented in this dissertation and the artifacts repository (see Section 3.7).

I used the categorization of vulnerabilities provided by Ghera as a coarse means to identify the vulnerabilities to evaluate each tool. If vulnerabilities were mis-categorized in Ghera, then I may have evaluated a tool against an inapplicable vulnerability or not evaluated a tool against an applicable vulnerability. This threat can be addressed by verifying Ghera's categorization of vulnerabilities or individually identifying the vulnerabilities that a tool is applicable to.

External Validity The above observations are based on the evaluation of 14 vulnerability detection tools. While this is a reasonably large set of tools, it may not be representative of the population of vulnerability detection tools. For example, it does not include commercial tools, it does not include free tools that failed to build. So, these observations should be considered as is only in the context of tools similar to the evaluated tools and more exploration should be conducted before generalizing the observations to other tools.

In a similar vein, the above observations are based on 42 known vulnerabilities that have been discovered by the community since 2011 and captured in Ghera since 2017. While this is a reasonably large set of vulnerabilities and it covers different capabilities of Android apps/platforms, it may not be representative of the entire population of Android app vulnerabilities. For example, it does not include different manifestations of a vulnerability stemming from the use of similar yet distinct APIs, it does not include vulnerabilities stemming from the use of APIs (e.g., graphics, UI) that are not part of the considered API categories, it does not include native vulnerabilities. So, these observations can be considered as is only in the context of the considered 42 known vulnerabilities. The observations can be considered with caution in the context of different manifestations of these vulnerabilities. More exploration should be conducted before generalizing the observations to other vulnerabilities.

4.6 Evaluation Artifacts

Ghera benchmarks used in the evaluations described in this manuscript are available at `https://bitbucket.org/secure-it-i/android-app-vulnerability-benchmarks/src?at=RekhaEval-3`.

A copy of specific versions of offline tools used in tools evaluation along with tool output from the evaluation are available in a publicly accessible repository: `https://bitbucket.org/secure-it-i/may2018`. Specifically, *vulevals* and *secevals* folders in the repository contain artifacts from the evaluation of vulnerability detection tools using benign apps and secure apps from Ghera, respectively. The repository also contains scripts used to automate the evaluation along with the instructions to repeat the evaluation.

To provide a high-level overview of the findings from tools evaluation, I have created a simple online dashboard of the findings at `https://secure-it-i.bitbucket.io/rekha/dashboard.html`. The findings on the dashboard are linked to the artifacts produced by each tool in the evaluation. I hope the dashboard will help app developers identify security tools that are well suited to check their apps for vulnerabilities and tool developers assess how well their tools fare against both known and new vulnerabilities and exploits.

4.7 Related Work

Android security has generated considerable interest in the past few years. This is evident by the sheer number of research efforts exploring Android security. Sufatrio et al. [21] summarized such efforts by creating a taxonomy of existing techniques to secure the Android ecosystem. They distilled the state of the art in Android security research and identified potential future research directions. While their effort assessed existing techniques theoretically on the merit of existing reports and results, this effort evaluated existing tools empirically by executing them against a common set of benchmarks; hence, these efforts are complementary.

In 2016, Reaves et al. [49] systematized Android security research focused on application analysis by considering Android app analysis tools that were published in 17 top venues since 2010. They also empirically evaluated the usability and applicability of the results of 7 Android app analysis tools. In contrast, this study evaluated 14 tools that detected vulnerabilities. Further, they used benchmarks from DroidBench, six vulnerable real-world apps, and top ten financial apps from Google Play store as inputs to tools. While the vulnerable real-world apps were likely authentic (i.e., they contained vulnerabilities), this was not the case with the financial apps or the DroidBench benchmarks. In contrast, all Ghera benchmarks used as inputs in this evaluation were authentic. While DroidBench focuses on ICC related vulnerabilities and use of taint analysis for vulnerability detection, Ghera is agnostic to the techniques underlying the tools and contains vulnerabilities related to ICC and other features of Android platform (e.g., crypto, storage, web). While their evaluation focused on the usability of tools (i.e., how easy is it to use the tool in practice? and how well does it work in practice?), this evaluation focused more on the effectiveness of tools in detecting known vulnerabilities and malicious behavior and less on the usability of tools. Despite these differences, the effort by Reaves et al. is closely related to this work.

Sadeghi et al. [63] conducted an exhaustive literature review of the use of program analysis techniques to address issues related to Android security. They identified trends, patterns, and gaps in existing literature along with the challenges and opportunities for future research. In comparison, this evaluation also exposes gaps in existing tools. However, it does so empirically while being agnostic to techniques underlying the tools (i.e., not limited to program analysis).

More recently, Pauck et al. [52] conducted an empirical study to check if Android static taint analysis tools keep their promises . Their evaluation uses DroidBench, ICCBench, and DIALDroidBench as inputs to tools. Since the authenticity of the apps in these benchmark suites was unknown, they developed a tool to help them confirm the presence/absence of vulnerabilities in the apps and used it to create 211 authentic benchmarks. Likewise, they created 26 authentic benchmarks based on the real-world apps from DIALDroidBench. Finally, they empirically evaluated the effectiveness and scalability of 6 static taint analysis tools using these 237 benchmarks. While their evaluation is similar to this evaluation in terms of the goals — understand the effectiveness of tools, there are non-trivial differences in the approaches and findings. First, unlike their evaluation, this evaluation used Ghera benchmarks which are demonstrably authentic and did not require extra effort to ensure authenticity as part of the evaluation. Further, while their evaluation is subject to bias and incompleteness due to manual identification of vulnerable/malicious information flows, this evaluation does not suffer from such aspects due to the intrinsic characteristics of Ghera benchmarks (e.g., tool/technique agnostic, authentic). Second, while they evaluated six security analysis tools, I evaluated 14 vulnerability detection tools (21 variations in total; see Table 4.4) including three tools evaluated by Pauck et al.. Further, while they evaluated only academic tools, this assessment considered academic tools and non-academic tools. Third, while their evaluation focused on tools based on static taint analysis, this evaluation was agnostic to the techniques underlying the tools. Their evaluation was limited to ICC related vulnerabilities while our evaluation covered vulnerabilities related to ICC and other features of the Android platform (e.g., crypto, storage, web). Fourth, while their evaluation used more than 200 synthetic apps and 26 real-world apps, this evaluation used only 84 synthetic apps (i.e., 42 vulnerable apps and 42 secure apps). However, since each benchmark in Ghera embodies a unique vulnerability, this evaluation is based on 42 unique vulnerabilities. In contrast, their evaluation is not based on unique vulnerabilities as not every DroidBench benchmark embodies a unique vulnerability (e.g., privacy leak due to constant index based array access vs privacy leak due to calculated index based array access).

4.8 Conclusion

When I started the evaluation, I expected many Android app security analysis tools to detect many of the known vulnerabilities. The reasons for this expectation was 1) there has been an explosion of efforts in recent years to develop security analysis tools and techniques for Android apps and 2) almost all of the considered vulnerabilities were discovered and reported before most of the evaluated tools were last developed/updated.

Contrary to my expectation, the evaluation suggests that most of the tools and techniques are able to independently detect only a small number of considered vulnerabilities. Further, all tools together are unable to detect all of the considered vulnerabilities.

These observations suggest if current and new security analysis tools and techniques are to be helpful in securing Android apps, then they need to be more effective in detecting vulnerabilities; specifically, starting with known vulnerabilities as a large portion of real-world apps use APIs associated with these vulnerabilities. A two-step approach to achieve this is 1) build and maintain an open, free, and public corpus of known Android app vulnerabilities in a verifiable and demonstrable form and 2) use the corpus to continuously and rigorously evaluate the effectiveness of Android app security analysis tools and techniques.

Chapter 5

SeMA: A Development Methodology to Secure Android Apps

Android apps have become an integral aspect of modern-day living. With the growing use of these apps, it is vital to secure them. Existing approaches to secure Android apps are curative, i.e., they are aimed at detecting vulnerabilities after they are introduced. Identifying and fixing vulnerabilities after they occur increase the cost of development. Consequently, there has been a growing call to integrate security into every phase of the software development life-cycle. In a 2014 study, Green and Smith argued that to build secure software, developers need support in various areas ranging from safer programming languages to better security testing tools [41]. Therefore, given the current landscape of Android app security and to support the call for secure software development, there is scope for exploring an alternative approach aimed at *preventing* vulnerabilities from occurring as opposed to *curing* them. In this chapter, I introduce SeMA, a development methodology based on existing design artefacts that can be used to build apps with verifiable security guarantees.

5.1 Motivation

In the last decade, researchers have developed a plethora of tools and techniques to help detect vulnerabilities in Android apps. Even so, apps with vulnerabilities find their way to app stores because app developers do not use these tools, or the tools are ineffective. The latter reason is supported by the findings presented in Chapter 4. Additionally, Pauck et al. [52] assessed six prominent taint analysis tools aimed at discovering vulnerabilities in Android and found tools to be ineffective in detecting vulnerabilities in real-world apps.

Existing approaches to secure Android apps are *curative* (i.e., detect vulnerabilities after they occur). Identifying and fixing a vulnerability in a curative manner increases the cost of development [94]. Therefore, given the current landscape of Android app security, I am proposing a preventive approach that can help prevent the occurrence of vulnerabilities in apps (as opposed to curing apps of vulnerabilities).

SeMA is based on an existing mobile app design technique called *storyboarding*. It treats security as a first-class citizen in the design phase and enables analysis and verification of security properties in an app's design. I demonstrate that SeMA can help prevent 49 of the 60 vulnerabilities captured in the Ghera benchmark suite (described in Chapter 2), which is more than the vulnerabilities collectively detected by tools evaluated in Chapter 4. Further a usability study with ten professional developers shows that SeMA helps reduce the time to develop an app and enables developers to prevent known vulnerabilities in their apps.

5.2 Background

SeMA borrows heavily from Model-Driven Development (MDD) and UX design techniques for Android apps.

Model Driven Development (MDD) In MDD, software is developed by iteratively refining abstract and formal models [37]. A model is meant to capture the application's behavior and is expressed in a domain-specific language (DSL). Apart from models, the domain-specific platform is a crucial entity in MDD. The domain-specific platform provides frameworks and APIs to enable easy refinement of a model into a platform-specific implementation. Since every aspect of an application can seldom be specified in the DSL, the resulting implementation is often extended with additional code; mostly, the business logic of the application.

Today, numerous tools exist to enable MDD in various domains. For example, Amazon uses TLA+ to develop web services [95, 96]. Tools like Alloy [97], UML/OCL [98], and UMLSec [99] help create and analyze models of software behavior, which form the basis for further development.

Similar approaches have been explored to simplify the development of mobile apps and reduce technical complexity and development costs. For example, Heitkotter et al. [100] developed MD^2, an MDD framework for making cross-platform apps. In this framework, a developer can describe an app is a platform-independent textual DSL and eventually generate platform-specific source code from the specification. Brambilla and others [101, 102] extended the Information Flow Modeling Language (IFML), a standard for depicting UI behavior, to enable the specification of a mobile app's GUI in a platform-independent manner. MobML is a collaborative framework for the design and development of data-intensive applications [103]. It offers four modeling languages, each addressing a different aspect of a mobile app (e.g., Navigation, UI, Content, and Business Logic), along with a code generator that translates the models into source code for the target platform. Vaupel et al. [104] propose a model-driven approach for developing mobile business apps that support the configuration of user role variants. In this approach, app variants are generated for each user role, which are then configured. Mobile Apps Generator or MAGS is a UML-based methodology that enables MDD for mobile apps [105]. It enables the specification of an app's requirements, structure, and behavior via use case diagrams, UML class diagrams, and UML state machines. Further, it generates a platform-specific implementation based on the UML models. Francese and others propose modeling an app's data-flow, control-flow, and user interactions in a finite state machine [106] and translating the finite state machine to non-native code that can be further modified by developers using cross-platform app development frameworks such as PhoneGap. AXIOM is a model-driven methodology that uses the Abstract Model Tree (AMT) representation for modeling platform-independent app behavior and requirements [107]. This representation forms the basis for code generation.

In addition to the academic efforts described above, a few commercial efforts have also explored MDD in the mobile app development space. For instance, the Mendix App Platform allows developers to visually describe various app components and execute those models in a runtime environment [108]. The IBM Rational Rhapsody provides developers with a visual representation of the Android framework API [109]. Developers can use these representations to create class diagrams of an app and execute them in a runtime environment.

Security Requirements Engineering Since MDD advocates developing software through iterative modeling, we could consider security requirements in MDD. This idea is very similar to the framework proposed by Hayley et al. [110] to elicit and analyze security requirements. The framework offers capabilities to define security requirements, context, and assumptions of a system along with capabilities to validate the requirements via formal and informal structured arguments. Further, this idea has been explored by recent approaches to reason about security of systems. For example, Basin et al. [111] proposed SecureUML for formally specifying access control requirements. They used these models to generate security architectures for distributed applications. Secure Tropos is a software development methodology that combines requirements engineering concepts with security-related concepts to aid the design and development of secure software systems [112]. DIGS is a framework to help requirements analysts define security goals and verify the completeness and correctness of the requirements w.r.t the defined goals [113].

Storyboarding Android app development teams use storyboarding to design an app's navigation [114, 115]. A storyboard is a sequence of screens and transitions between the screens. It serves as a model for the user's observed behavior in terms of screens and transitions. Numerous tools such as Xcode [116], Sketch [117], and Android's JetPack [118] help express a storyboard digitally. The storyboarding process is *participatory* in nature because it allows designers to get feedback from potential users about the features captured in the storyboard and from developers about the feasibility of implementing those features. However, traditional storyboards cannot capture an app's behavior (beyond UI and navigational possibilities). This limitation of storyboards hinders collaboration.

MDD with Storyboarding Existing MDD approaches are based on software architecture/design artifacts (e.g., UML diagrams) [37, 111, 112, 119]. While storyboards are structurally similar to these artifacts, storyboards cannot capture app behaviors that are crucial to enable model driven development of mobile apps. To remedy this situation, storyboards can be extended with capabilities to capture an app's behavior and then used as models of apps.

This adaptation offers numerous benefits. First, since storyboards are now an integral part of mobile app development IDEs [116, 118], storyboard based MDD can seamlessly fit into the existing mobile app development life cycle. Second, an extended storyboard can serve as a common substrate for collaboration between designers and developers to specify an app's behavior along with its UI and navigational features. Third, an extended storyboard can serve as a basis for formal analysis of an app's behavior. The abstractness of the models helps with the analysis because analyzing behaviors captured in an abstract model is relatively easier than extracting and analyzing the behaviors captured in code. Finally, the storyboard can serve as a reference for an app's behavior when auditing the app's implementation.

5.3 The Methodology

The proposed methodology, SeMA, enables the reasoning and verification of security properties of an app's storyboard via iterative refinement. The development process of SeMA is shown in Figure 5.2. The process begins with a developer sketching the initial storyboard of an app as shown in Figure 5.1. The developer then extends the storyboard with the app's behavior and checks if the behaviors satisfy various pre-defined security properties. The

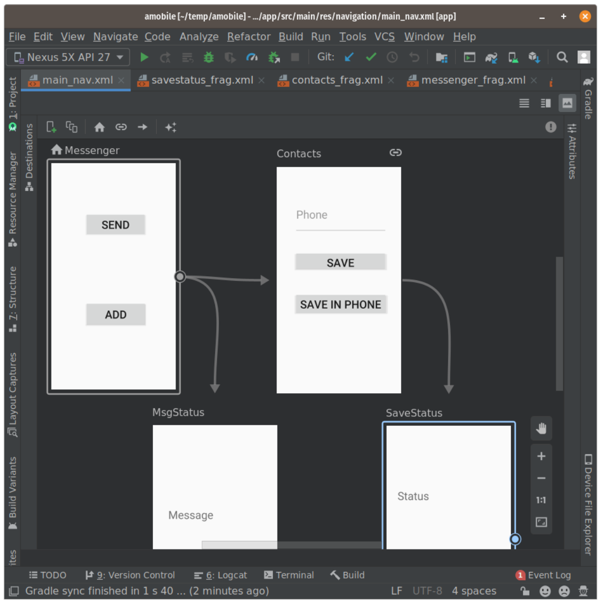


Figure 5.1: *Screenshot of the initial storyboard in Android Studio*

developer may repeat the previous steps to revise and refine the behaviors. Once the storyboard has captured the behavior as intended by the developer while satisfying pre-defined security properties, the developer generates an implementation from the storyboard with a push of a button. As the final step, the developer adds business logic to the implementation via hooks provided in the generated code.

5.3.1 Extended Storyboard

A *traditional storyboard* used in the design of mobile apps is composed of screens and transitions between screens. A screen is a collection of named widgets that allow the user to interact with the app (e.g., clicking a button). A transition (edge) between two screens depicts a navigation path from the source screen to the destination screen. The basic structure of a storyboard defines the navigational paths in an app.

For example, Figure 5.3 shows the traditional storyboard of an app with four screens: *Messenger*, *Contacts*, *MsgStatus*, and *SaveStatus*. Starting from the *Messenger* screen, a user can either add contact numbers to the app via *Contacts* screen or send a message to all saved contact numbers.

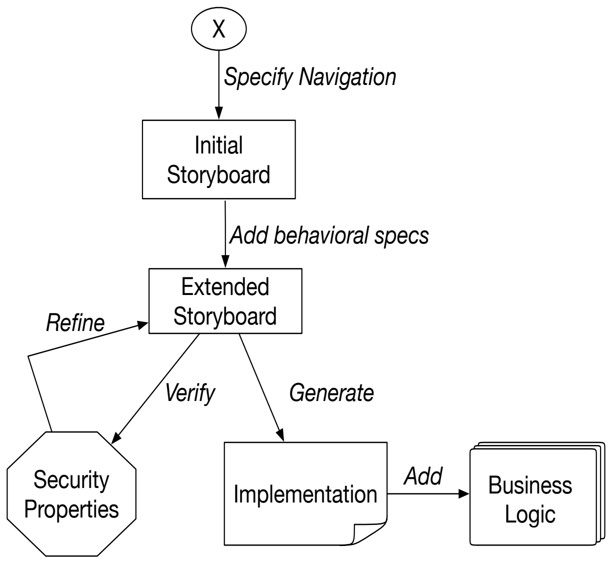


Figure 5.2: *A schematic of the steps in SeMA*

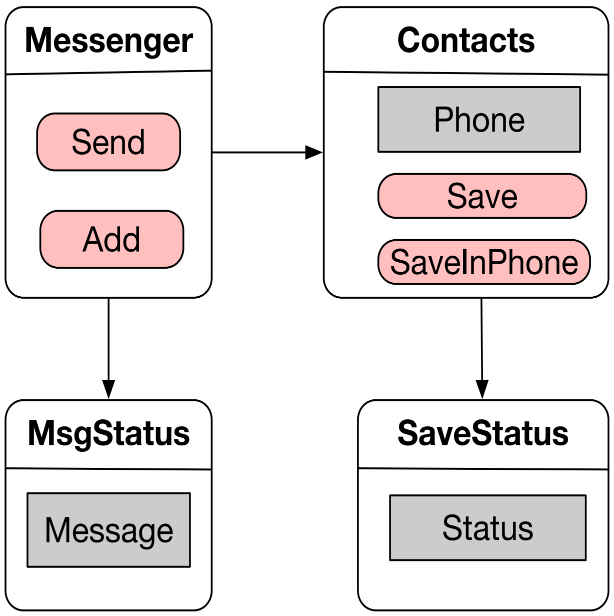


Figure 5.3: *Diagrammatic representation of the initial storyboard*

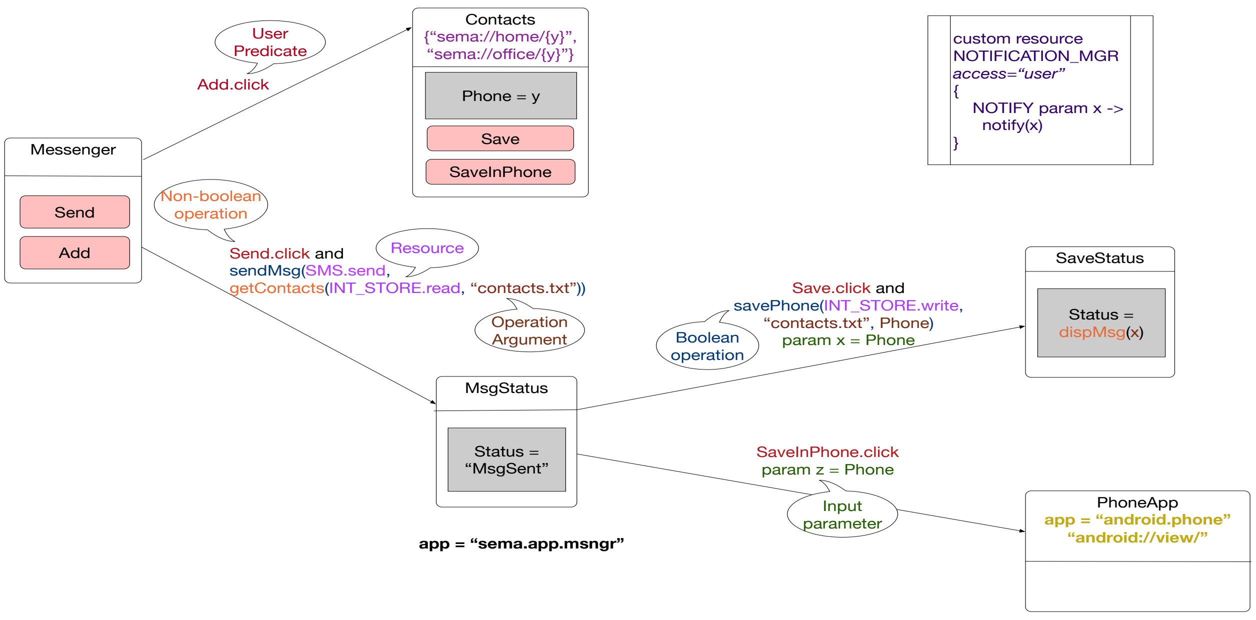

User Predicate
Add.click
Contacts
{"sema://home/{y}", "sema://office/{y}"}
Phone = y
Save
SaveInPhone
custom resource NOTIFICATION_MGR access="user"
{
NOTIFY param x -> notify(x)
}
Messenger
Send
Add
Non-boolean operation
Send.click and sendMsg(SMS.send, getContacts(INT_STORE.read, "contacts.txt"))
Resource
Operation Argument
Save.click and savePhone(INT_STORE.write, "contacts.txt", Phone) param x = Phone
Boolean operation
SaveStatus
Status = dispMsg(x)
MsgStatus
Status = "MsgSent"
SaveInPhone.click param z = Phone
Input parameter
PhoneApp
app = "android.phone"
"android://view/"
app = "sema.app.msngr"

Figure 5.4: *Example of an extended storyboard. The bubbles are not part of the storyboard. They help the reader understand the storyboard entities.*

A traditional storyboard does not support the specification of app behavior. Hence, I propose the following extensions to traditional storyboards to enable the specification of app behaviors in storyboards (i.e., enrich a traditional storyboard as in Figure 5.3 into an extended storyboard as in Figure 5.4).

App Identity Every storyboard has an *app* attribute, which is a unique string constant used to identify the app described by the storyboard.

Extensions to Screens Screens are extended with a mandatory *name*, optional *input parameters* (in green in Figure 5.4), and optional *URIs* (in purple in Figure 5.4).

Input parameters of a screen are similar to parameters of a function in mainstream programming. Input parameters bind to the values (arguments) provided when the screen is activated by either another screen in the app via an incoming transition or an app via the screen's *URI*.

A *URI* associated with a screen can be used to access the screen from outside the app. A URI can have input parameters; similar to query parameters in web URLs. URI input parameters serve as input parameters of the screen. All URIs associated with a screen must have the same set of input parameters. For example, both URIs associated with the *Contacts* screen in Figure 5.4, have the input parameter *y*. External apps accessing a screen via its URI must provide the arguments corresponding to the URI's input parameters. Every URI without its parameters must be unique in an app.

Proxy Screens of External Apps Apps often interact with external apps. To capture this interaction, *proxy screen* representing a screens of an external app can be included an extended storyboard of an app; see *PhoneApp* in Figure 5.4. Such proxy screens have a mandatory *name*, a mandatory *URI*, and an optional *app* attributes. If *app* is specified in a proxy screen, then the proxy screen denotes the screen identified by the *URI* in the app named *app*. If *app* is not specified, then the proxy screen denotes any screen identified by the *URI* in an app installed on the device and determined by Android.

Extensions to Widgets Widgets are extended with a mandatory *value* that can be assigned by the developer (e.g., in labels), entered by the user (e.g., in fields), provided by an input parameter of the containing screen (e.g., when the screen is activated by a transition), or returned by an operation. Based on the displayed content, widgets can be of different types (e.g., a label and text widget display plain text while a web widget displays web content). Further, depending on the widget's type, a widget can be configured with a pre-defined set of rules that regulate the data displayed in a widget. For example, a *WebView* widget can be configured with a whitelist of trusted URL patterns (via *trusted-patterns* attribute) to display content only from URLs in the whitelist.

Resources Android provides apps with resources with different capabilities (e.g., storage, networking). Hence, to complement this aspect, storyboards are extended with a pre-defined set of resources with specific capabilities that can be used by the apps being designed. [1] Android apps can offer UI-less services to other apps (e.g., broadcast receivers, content providers). Such services are denoted by custom resources in storyboards. A *custom resource*

[1] The current realization of SeMA supports a subset of pre-defined resources offered by Android.

offers capabilities that can be used by apps installed on the device. Each custom resource has a mandatory identifier that is unique to the app. Each capability of a resource has a mandatory identifier that is unique to the resource. Also, each capability that has security implications can be marked as *privileged*. Access to a resource and its capabilities can be controlled via the *access* attribute of the resource. This attribute can take on one of the following three values: *all* implying any app can be accessed the resource/capability, *user* implying user must grant a specific permission to access the resource/capability, and *own* implying only the resource defining app x or an app that shares the digital signature of app x. For example, in Figure 5.4, NOTIFICATION_MGR is a custom resource that offers NOTIFY capability with a *notify* operation. Based on its access attribute, a client will need to seek the user's permission to use its NOTIFY capability.

Operations In an extended storyboard, an *operation* indicates a task to be performed (e.g., read from a file, get contents from a web server). An operation has a *name*, returns a value, may have *input parameters*, and may use a *capability* (provided by a resource). An operation is used by mentioning its name along with arguments and any required capabilities. For example, in Figure 5.4, operation *savePhone* is used to save data in the device's internal storage by using the *write* capability of the internal storage device exposed as the resource *INT_STORE*. A use of an operation introduces (declares) it in the storyboard. An operation is defined in the generated implementation (described in Section 5.3.5). Use of operations *must* be consistent (i.e., a non-boolean operation cannot be used in a boolean value position).

Extension to Transitions Transitions between screens can be adorned with constraints that when satisfied enable/trigger a transition. A constraint is a conjunction of a *user action* (e.g., click of a button) and a set of *boolean operations*. A constraint is satisfied when the user action is performed (e.g., *save* button is clicked) and every boolean operation evaluates to true. For example, in Figure 5.4, the transition from *Contacts* screen to *SaveStatus* screen is taken only when the user clicks the *Save* button and the *savePhone* operation evaluates to *true*, that is, the value of *Phone* is successfully saved in "contacts.txt", a file on the internal storage of the device.

As part of a transition, arguments are provided to the input parameters of the destination screen (in green 5.4). An argument can be a literal specified in the storyboard, a value available in the source screen of the transition (e.g., value of a contained widget, input parameter to the screen), or a value returned by an operation. Further, every transition to a screen must provide values (arguments) for every input parameter of that screen. For example, if there are two transitions *t1* and *t2* to screen s with input parameters x and y, then arguments for both x and y must be provided along both *t1* and *t2*.

Multiple outgoing transitions from a screen may be simultaneously enabled when their constraints are not mutually exclusive. Hence, to handle such situations, all outgoing transitions from a screen must be totally ordered; see *Contacts* screen in Figure 5.4. The implementation derived from the storyboard will evaluate the constraints according to the specified order of transitions and take the first enabled transition.

5.3.2 A Formal Specification of SeMA

This section formally defines the syntax and semantics of the extended storyboard, along with progress and safety properties guaranteed by the semantics.

Syntax

The syntax domains and the meta-variables that range over the syntax domains used to specify the storyboard language's formal syntax are defined in Table 5.1.

Assume that the syntactic structure of values, identifiers, names, and URI strings is given. For example, identifiers consist of non-empty string of letters. The remaining syntactic sets are defined inductively via formation rules shown below.

The meta-variables can be sub-scripted or primed e.g. s', s_0 stands for an element in the set S. The symbol ? is used to denote an *optional* syntactic construct. The formation rules are presented in a variant of BNF (Backus-Naur form) in Figure 5.5.

$bv \in$ BV, BV $= \{true, false\}$	$v \in$ V, non-boolean values
$app \in$ APP, apps	$appid \in$ APPID, app identifiers
$vid \in$ VID, screen parameter identifiers	$wid \in$ WID, widget identifiers
$sid \in$ SID, screen identifiers	$a \in$ ARG, operation arguments
$s \in$ S, screens	$ps \in$ PS, proxy screens
$w \in$ W, widgets	$wt \in$ WT, widget types
$tr \in$ TR, transitions	$b \in$ B, boolean expressions
$r \in$ R, resources	$rn \in$ RN
$f \in$ F, operations	$p \in$ P, screen parameters
$fn \in$ FN, operation identifiers	$tid \in$ TID, transition identifiers
$u \in$ U, screen URIs	$us \in$ US, URI strings
$c \in$ C, resource capabilities	$cn \in$ CN, capability names
$g \in$ G, gestures.	

Table 5.1: *Syntax Domain*

app ::= $appId$ s r?

s ::= **screen** sid u? w tr? | **proxy safe**? sid $appId$? us

w ::= **safe**? wt wid (v | f | vid) | $w_0 w_1$

wt ::= **TextView** | **EditText** | **Button** | **WebView**

f ::= **fun** fn ($rn.cn$)? a?

r ::= (**access all** | **user** | **own**) resource rn c | $r_0 r_1$

c ::= **priv**? cn f | $c_0 c_1$

a ::= (**safe**? v | vid | wid | f) | $a_0 a_1$

tr ::= **transition** tid **dest** sid **cond** ua (**and** b)? p? | $tr_0 tr_1$

ua ::= $wid.g$

g ::= **click** | **swipe** | **drag**

p ::= **param** vid (wid | vid_0 | v | f) | $p_0 p_1$

u ::= us | us/k | $u_0 u_1$

k ::= vid | $k_0 k_1$

b ::= bv | f | b_0 **and** b_1 | b_0 **or** b_1 | **not** b

Figure 5.5: *Formation Rules*

Semantics

I present the small-step operational semantics of a storyboard in SeMA. Before, describing the semantics, I introduce the function and operators in the meta-language that will be needed to understand the semantics.

1. $\phi : ID \rightarrow SID$, where $ID = VID \cup WID$ is a function that is used to keep track of the screen associated with a given widget or variable identifier. For a widget identifier $wid \in WID$, it returns a screen identifier *sid* where *wid* is a widget in *sid*. For a variable identifier $vid \in VID$, it returns a screen identifier *sid* such that *vid* as an input parameter of *sid*.

2. $gen(id, sid)$ is an operator that takes a pair $(id, sid) : id \in ID, sid \in SID$ and returns a new $id\prime \in ID$.

3. $\sigma : ID \rightharpoonup V$ is a partial function that maps IDs to Values and is used to denote the state of the app. This function keeps track of the values assigned to widget and variable identifiers in a screen.

4. $init : APPID \rightarrow SID$ is a function that maps an *appId* to the start screen identifier *sid* of an app. The start screen of an app is the screen that the user sees when the app is started for the first time. This function returns the screen ID in an app that needs to be displayed when the app is started.

5. $eval : F \rightarrow X$, where $X = V \cup BV$ is a function that returns a *non-boolean value* or a *boolean value* for an operation $f \in F$. This function interprets an operation used in the storyboard.

6. $evalUriVar : ID \rightharpoonup V$ is a partial function that returns the value associated with a variable identifier associated with the URI of a screen. The value is provided by an external app that uses the URI to trigger the corresponding screen. If no external app provides a value for a variable identifier *id*, then $evalUriVar(id) = \perp$.

7. $\mathit{outTr} : SID \rightarrow TR$ is a function that returns all outgoing transitions $tr \in TR$ from a screen $sid \in SID$.

8. $\mathit{occur} : UA \rightarrow BV$ is a predicate that is *true* if the user performed a gesture (e.g., button click) and is *false* if the gesture was not performed. This predicate is used to capture any gestures a user may perform on the widgets in a screen visible to the user.

9. $\mathit{validScr} : (SID) \rightarrow BV$ is a predicate that is *true* if a given $sid \in SID$ is the ID of a screen in the app; and *false* otherwise.

10. $\mathit{compr} : (h, h') \rightarrow BV$ is a predicate that is *true* if $\forall x \in \mathit{dom}(h) : (h(x) = v \wedge h'(x) =\perp) \vee (h(x) =\perp \wedge h'(x) = v)$; *false* otherwise.

11. $\mathit{stop} : (APPID) \rightarrow BV$ is a predicate that is *true* if an app with $\mathit{appId} \in APPID$ is moved to the background or is killed explicitly by the user or Android; and *false* otherwise.

12. $\rho : (RN, CN) \rightarrow BV$ is a predicate that is *true* if a capability identifier $cn \in CN$ offered by a resource identifier $rn \in RN$ is defined; and *false* otherwise.

13. $\mathit{erase}(h, h')$ is a binary operator that takes two partial functions h,h' and returns another partial function g such that $\forall x \in \mathit{dom}(h') : g(x) = h'(x)$, but $g(x) =\perp$ when $h(x) \neq\perp \wedge h'(x) \neq\perp$. The operator is required to modify the state of the app, σ, when the transition to a screen is taken.

14. $\mathit{erase}_K(h, h\prime)$ is a binary operator that takes two partial functions h,h' and returns another partial function g such that $\forall x \in K, K \subseteq \mathit{dom}(h') : g(x) = h'(x)$, but $g(x) =\perp$ when $h(x) \neq\perp \wedge h'(x) \neq\perp$. The operator is used to change the state of the app, σ, when a transition from a screen to itself is taken.

In addition to the meta-functions and meta-operators, I assume that the custom resources defined in a storyboard are parsed beforehand and stored as the app's custom resources. Hence, resource definitions are not explicitly described in the semantics presented below.

$$\frac{init(appId) = sid}{\langle appId\, s \;\; r, (\bot, \sigma)\rangle \rightarrow \langle appId\, s \;\; r, (sid, \sigma)\rangle} \tag{5.1}$$

$$\frac{stop(appId) = true}{\langle appId\, s \;\; r, (sid, \sigma)\rangle \rightarrow (\bot, erase(\sigma, \sigma))} \tag{5.2}$$

$$\frac{stop(appId) = false \qquad \langle s, (sid, \sigma)\rangle \rightarrow (sid', \sigma')}{\langle appId\, s \;\; r, (sid, \sigma)\rangle \rightarrow \langle appId\, s \;\; r, (sid', \sigma')\rangle} \tag{5.3}$$

Table 5.2: *App-related Semantic Rules.*

Every syntactic construct defined in Section 5.3.2 is evaluated in a configuration of the form (sid, σ), where sid denotes the *current screen* visible to the user of the app and σ denotes the state of the app when the user is at screen sid. The *initial configuration* of the app $(\bot, \sigma)$.

App-related rules The rules listed in Table 5.2 describe an app's behavior when a user starts an app, interacts with the app, or closes the app.

When an app is started, as per rule 5.1, it moves from its initial configuration to a configuration where the current screen is set to $sid \in SID$. sid is obtained from the meta-function *init*.

As per rule 5.2, if an app is stopped by the user or Android, then the terminal configuration $(\bot, \sigma)$, such that $\forall x \in dom(\sigma) : \sigma(x) = \bot$, is reached. This configuration indicates that no screen of the app is visible to the user and the state of the app is undefined.

As per rule 5.3, an app in a configuration (sid, σ) moves to a new configuration if a screen sid in the app changes the configuration.

Screen-related rules. The rules listed in Table 5.3 apply to an app's screens. They are informally described in Section 5.3.1 as screen extensions.

Rule 5.4 is applicable when the app is in a configuration (sid, σ) and sid is the identifier

$$\frac{\langle s_0, (sid, \sigma)\rangle \rightarrow (sid', \sigma')}{\langle s_0 s_1, (sid, \sigma)\rangle \rightarrow (sid', \sigma')} \tag{5.4}$$

$$\frac{\langle s_1, (sid, \sigma)\rangle \rightarrow (sid', \sigma')}{\langle s_0 s_1, (sid, \sigma)\rangle \rightarrow (sid', \sigma')} \tag{5.5}$$

$$\frac{\langle w, (sid, \sigma)\rangle \rightarrow (sid, \sigma') \qquad \langle tr, (sid, \sigma')\rangle \rightarrow (sid', \sigma'')}{\langle \mathbf{screen}\ sid\ u\ w\ tr, (sid, \sigma)\rangle \rightarrow (sid', \sigma'')} \tag{5.6}$$

$$\frac{\langle w, (sid, \sigma)\rangle \rightarrow (sid, \sigma')}{\langle \mathbf{screen}\ sid\ u\ w\ , (sid, \sigma)\rangle \rightarrow (sid, \sigma')} \tag{5.7}$$

$$\langle \mathbf{proxy}\ sid\ appId\ u, (sid, \sigma)\rangle \rightarrow (sid, \sigma) \tag{5.8}$$

Table 5.3: *Screen-related Semantic Rules.*

of the first screen s_0 in the sequence $s_0 s_1$.

Rule 5.5 is applicable when the app is in a configuration (sid, σ) and sid is the identifier of some screen in the sequence $s_0 s_1$.

As per rule 5.6, an app moves from current screen sid with state σ to a screen $sid\prime$ with state $\sigma\prime\prime$ when the widgets in the screen sid extend the state σ to $\sigma\prime$ and one of the outgoing transitions sets the current screen to $sid\prime$ with new state $\sigma\prime\prime$.

As per rule 5.7, a screen with no outgoing transitions causes the current state σ to change to a new state $\sigma\prime$ when the widgets in the screen sid extend the state σ to $\sigma\prime$.

A proxy screen does not result in any configuration change as per rule 5.8.

Widget-related rules The rules listed in Table 5.4 correspond to linking widgets to data sources in a screen. These rules are informally described as *Widget Extensions* in Section 5.3.1.

As per rule 5.9, a sequence of widgets in a screen sid extends the corresponding state σ

$$\frac{\langle w_0, (sid, \sigma)\rangle \rightarrow (sid, \sigma') \qquad \langle w_1, (sid, \sigma')\rangle \rightarrow (sid, \sigma'')}{\langle w_0 w_1, (sid, \sigma)\rangle \rightarrow (sid, \sigma'')} \tag{5.9}$$

$$\frac{\langle x, (sid, \sigma)\rangle \rightarrow v}{\langle wt\ wid\ x, (sid, \sigma)\rangle \rightarrow (sid, \sigma[gen(wid, sid) \mapsto v])}, x ::= vid\ or\ f \tag{5.10}$$

$$\frac{\langle vid, (sid, \sigma)\rangle \rightarrow \perp}{\langle wt\ wid\ vid, (sid, \sigma)\rangle \rightarrow (sid, \sigma)} \tag{5.11}$$

$$\langle wt\ wid\ v, (sid, \sigma)\rangle \rightarrow (sid, \sigma[gen(wid, sid) \mapsto v]) \tag{5.12}$$

Table 5.4: *Widget-related Semantic Rules.*

to σ'' if each widget in the sequence extends σ.

As per rule 5.10, a widget *wid* extends state σ with value v if *wid* is initialized with a variable *vid* or an operation f, and *vid* or f evaluates to v in state σ.

As per rule 5.11, a widget *wid* does not change the current configuration if *wid* is initialized with an invalid variable (i.e., the variable is undefined in σ).

As per rule 5.12, a widget *wid* extends state σ with value v if *wid* is initialized with the value v.

Transition-related rules The rules listed in Table 5.5 specify an app's behavior in the context of outgoing transitions from a screen. They are informally described as *Extensions to Transition* in Section 5.3.1.

As per rule 5.13, if the first transition in an ordered set of transitions is taken, then the remaining transitions are not evaluated.

As per rule 5.14, if the first transition in an ordered set of transitions is not taken, then the remaining transitions in the ordered set are evaluated.

As per rule 5.15, a transition from a screen to a different screen is taken if the user action associated with it evaluates to *true*, the associated boolean condition evaluates to *true*, and

$$\frac{\langle tr_0, (sid, \sigma)\rangle \rightarrow (sid', \sigma') \qquad compr(\sigma, \sigma')}{\langle tr_0 tr_1, (sid, \sigma)\rangle \rightarrow (sid', \sigma')} \tag{5.13}$$

$$\frac{\langle tr_0, (sid, \sigma)\rangle \rightarrow (sid, \sigma) \qquad \langle tr_1, (sid, \sigma)\rangle \rightarrow (sid', \sigma')}{\langle tr_0 tr_1, (sid, \sigma)\rangle \rightarrow (sid', \sigma')} \tag{5.14}$$

$$\frac{\begin{array}{c}\langle ua, (sid, \sigma)\rangle \rightarrow true \\ \langle b, (sid, \sigma)\rangle \rightarrow true \qquad validScr(sid') = true \qquad \langle p, (sid, \sigma)\rangle \rightarrow (sid, \sigma') \qquad sid \neq sid'\end{array}}{\langle \textbf{transition}\ tid\ \textbf{dest}\ sid'\ (\textbf{cond}\ ua\ \textbf{and}\ b)\ p, (sid, \sigma)\rangle \rightarrow (sid', erase(\sigma, \sigma'))} \tag{5.15}$$

$$\frac{\langle ua, (sid, \sigma)\rangle \rightarrow true \qquad \langle b, (sid, \sigma)\rangle \rightarrow true \qquad validScr(sid') = true}{\langle \textbf{transition}\ tid\ \textbf{dest}\ sid'\ (\textbf{cond}\ ua\ \textbf{and}\ b)\ , (sid, \sigma)\rangle \rightarrow (sid', erase(\sigma, \sigma'))} \tag{5.16}$$

$$\frac{\begin{array}{c}\langle ua, (sid, \sigma)\rangle \rightarrow true \\ \langle b, (sid, \sigma)\rangle \rightarrow true \qquad validScr(sid') = true \qquad \langle p, (sid, \sigma)\rangle \rightarrow (sid, \sigma')\end{array}}{\langle \textbf{transition}\ tid\ \textbf{dest}\ sid\ (\textbf{cond}\ ua\ \textbf{and}\ b)\ p, (sid, \sigma)\rangle \rightarrow (sid, erase_{WID}(\sigma, \sigma'))} \tag{5.17}$$

$$\frac{\langle ua, (sid, \sigma)\rangle \rightarrow false}{\langle \textbf{transition}\ tid\ \textbf{dest}\ sid\prime\ (\textbf{cond}\ ua\ \textbf{and}\ b)\ p, (sid, \sigma)\rangle \rightarrow (sid, \sigma)} \tag{5.18}$$

$$\frac{\langle b, (sid, \sigma)\rangle \rightarrow false}{\langle \textbf{transition}\ tid\ \textbf{dest}\ sid'\ (\textbf{cond}\ ua\ \textbf{and}\ b)\ p, (sid, \sigma)\rangle \rightarrow (sid, \sigma)} \tag{5.19}$$

Table 5.5: *Transition-related Semantic Rules.*

the arguments to the destination screen's parameters p extend the current state σ. Rule 5.16 is a variant of this rule for a transition without associated input parameters. Rule 5.17 is a variant of this rule for a transition from a screen to itself.

As per rule 5.18, a transition is not taken if the user action associated with it evaluates to *false*.

As per rule 5.19, a transition is not taken if the boolean condition associated with it evaluates to *false*.

Operation-related rules The rules listed in Table 5.6 are used to evaluate an operation used in the storyboard. These rules correspond to the informal description of *Operations* in Section 5.3.1.

As per rule 5.20, an operation f evaluates to a value v, if the meta-function *eval* evaluates f to a non-boolean value v and the resource used in the operation is defined. Rule 5.21 is a variant of this rule for operations with arguments.

As per rule 5.22, an operation f evaluates to a value v, if the meta-function *eval* evaluates f to a non-boolean value v. Rule 5.23 is a variant of this rule for operations without resources or arguments.

As per rule 5.24, an operation f evaluates to a boolean value bv, if the meta-function *eval* evaluates f to a boolean bv and the resource used in the operation is defined. Rule 5.25 is a variant of this rule for operation with arguments.

As per rule 5.26, an operation f evaluates to a boolean value bv, if the meta-function *eval* evaluates f to a boolean bv. Rule 5.27 is a variant of this rule for operations without resources or arguments.

ID-related rules The rules listed in Table 5.7 are used to evaluate a widget or variable identifiers.

As per rule 5.28, a widget identifier *wid* evaluates to a value v under state σ if the state σ maps *wid* to the value v.

As per rule 5.29, a gesture g on a widget *wid* evaluates to *true*, if the gesture occurs on

$$\frac{\rho(rn, cn) = true \qquad eval(f) = v}{\langle \mathbf{fun}\, fn\; rn.cn\,, (sid, \sigma)\rangle \rightarrow v} \tag{5.20}$$

$$\frac{\rho(rn, cn) = true \qquad eval(f) = v}{\langle \mathbf{fun}\, fn\; rn.cn\; a, (sid, \sigma)\rangle \rightarrow v} \tag{5.21}$$

$$\frac{eval(f) = v}{\langle \mathbf{fun}\, fn\;\; a, (sid, \sigma)\rangle \rightarrow v} \tag{5.22}$$

$$\frac{eval(f) = v}{\langle \mathbf{fun}\, fn\;\; , (sid, \sigma)\rangle \rightarrow v} \tag{5.23}$$

$$\frac{\rho(rn, cn) = true \qquad eval(f) = bv}{\langle \mathbf{fun}\, fn\; rn.cn\,, (sid, \sigma)\rangle \rightarrow bv} \tag{5.24}$$

$$\frac{\rho(rn, cn) = true \qquad eval(f) = bv}{\langle \mathbf{fun}\, fn\; rn.cn\; a, (sid, \sigma)\rangle \rightarrow bv} \tag{5.25}$$

$$\frac{eval(f) = bv}{\langle \mathbf{fun}\, fn\;\; a, (sid, \sigma)\rangle \rightarrow bv} \tag{5.26}$$

$$\frac{eval(f) = bv}{\langle \mathbf{fun}\, fn\;\; , (sid, \sigma)\rangle \rightarrow bv} \tag{5.27}$$

Table 5.6: *Operation-related Semantic Rules.*

$$\frac{\sigma(gen(wid, sid)) = v}{\langle wid, (sid, \sigma)\rangle \rightarrow v} \tag{5.28}$$

$$\frac{\langle occurs(wid, g), (sid, \sigma)\rangle \rightarrow bv}{\langle wid.g, (sid, \sigma)\rangle \rightarrow bv} \tag{5.29}$$

$$\frac{\sigma(gen(vid, sid)) = v}{\langle vid, (sid, \sigma)\rangle \rightarrow v} \tag{5.30}$$

$$\frac{\sigma(gen(vid, sid)) = \perp \qquad evalUriVar(gen(vid, sid)) = v}{\langle vid, (sid, \sigma)\rangle \rightarrow v} \tag{5.31}$$

$$\frac{\sigma(gen(vid, sid)) = \perp \qquad evalUriVar(gen(vid, sid)) = \perp}{\langle vid, (sid, \sigma)\rangle \rightarrow \perp} \tag{5.32}$$

$$\frac{\sigma(gen(wid, sid)) = \perp}{\langle wid, (sid, \sigma)\rangle \rightarrow \perp} \tag{5.33}$$

Table 5.7: *ID-related Semantic Rules.*

$$\frac{\langle p_0, (sid, \sigma)\rangle \rightarrow (sid, \sigma') \qquad \langle p_1, (sid, \sigma')\rangle \rightarrow (sid, \sigma'')}{\langle p_0 p_1, (sid, \sigma)\rangle \rightarrow (sid, \sigma'')} \tag{5.34}$$

$$\langle \textbf{param}\ vid\ v, (sid, \sigma)\rangle \rightarrow (sid, \sigma[gen(vid, \phi(vid)) \mapsto v]) \tag{5.35}$$

$$\frac{\langle x, (sid, \sigma)\rangle \rightarrow v}{\langle \textbf{param}\ vid_0\ x, (sid, \sigma)\rangle \rightarrow (sid, \sigma[gen(vid_0, \phi(vid_0)) \mapsto v])}, x ::= vid_1, wid,\ or\ f \tag{5.36}$$

$$\frac{\langle x, (sid, \sigma)\rangle \rightarrow \perp}{\langle \textbf{param}\ vid_0\ x, (sid, \sigma)\rangle \rightarrow (sid, \sigma)}, x ::= vid_1\ or\ wid \tag{5.37}$$

Table 5.8: *Screen's input parameter-related Semantic Rules.*

the widget; *false* otherwise.

As per rule 5.30, a variable identifier vid evaluates to a value v under state σ if the state σ maps vid to the value v.

As per rule 5.31, a variable identifier vid evaluates to a value v, if $\sigma(vid)$ is undefined, but an external app has provided the value v for vid via $evalUriVar$.

As per rule 5.32, a variable evaluates to $\perp$ if the variable identifier is undefined in σ and no external app has provided a value for the variable.

As per rule 5.33, a widget evaluates to $\perp$ if the widget identifier is undefined in σ.

Screen's input parameter-related rules The rules listed Table 5.8 are used to evaluate the arguments provided to input parameters of a screen as part of transitions to that screen. An informal description of these rules is provided in Section 5.3.1 as *Extensions to Transitions.*

As per rule 5.34, a sequence of syntactic constructs that provide arguments to the parameters of a screen sid extend the state σ, if each construct provides an argument to a parameter (i.e., each construct in the sequence extends the state σ).

As per rule 5.35, a syntactic construct that provides an argument to a screen's parameter extends the state σ in the current configuration with the value v, if the value v is provided as an argument to parameter ID *vid*.

As per rule 5.36, a syntactic construct that provides an argument to a screen's parameter extends the state σ in the current configuration with the value of x, if x has the value v in σ and x is either a variable, widget or an operation.

As per rule 5.37, A syntactic construct that provides an argument to a screen's parameter does not change the current configuration if the variable or widget that provides the argument is undefined in σ.

Boolean-related rules The rules listed in Table 5.9 are related to boolean expressions and are similar to traditional notions of conjunction, disjunction, and negation in logical statements.

Example

Let us assume that the storyboard of the app in Figure 5.4 is composed of a sequence of screens s_0s_1, where s_0 is the *Messenger* screen and s_1 is a sequence of the remaining screens. Further assume that when the user starts the app, *Messenger* screen is displayed. Hence, *init(appId)* = *Messenger*, where *appId* is the identifier (*"sema.app.msngr"*) of the app. Finally assume that there is an ordered set of outgoing transitions, tr_0tr_1, from *Messenger* screen, such that tr_0 is the transition from *Messenger* to *MsgStatus* and the tr_1 is an ordered set of the remaining transitions from *Messenger*. Given these assumptions, we will apply the semantic rules in the previous section to interpret a snippet of the storyboard as shown in Table 5.10.

Safety and Progress

I say that an app is safe if the app does not reach an *invalid configuration* or a *terminal configuration*. Since there are no invalid configurations, an app does not reach such a configuration. As far as terminal configuration is concerned, I define it as the $(\perp, \sigma)$, such that $\forall x \in dom(\sigma) : \sigma(x) = \perp$. The only time an app reaches the terminal configuration is

$$\frac{\langle b_0, (sid, \sigma)\rangle \rightarrow true \qquad \langle b_1, (sid, \sigma)\rangle \rightarrow true}{\langle b_0 \textbf{ and } b_1, (sid, \sigma)\rangle \rightarrow true} \tag{5.38}$$

$$\frac{\langle b_0, (sid, \sigma)\rangle \rightarrow false}{\langle b_0 \textbf{ and } b_1, (sid, \sigma)\rangle \rightarrow false} \tag{5.39}$$

$$\frac{\langle b_1, (sid, \sigma)\rangle \rightarrow false}{\langle b_0 \textbf{ and } b_1, (sid, \sigma)\rangle \rightarrow false} \tag{5.40}$$

$$\frac{\langle b_0, (sid, \sigma)\rangle \rightarrow true}{\langle b_0 \textbf{ or } b_1, (sid, \sigma)\rangle \rightarrow true} \tag{5.41}$$

$$\frac{\langle b_1, (sid, \sigma)\rangle \rightarrow true}{\langle b_0 \textbf{ or } b_1, (sid, \sigma)\rangle \rightarrow true} \tag{5.42}$$

$$\frac{\langle b_0, (sid, \sigma)\rangle \rightarrow false \qquad \langle b_1, (sid, \sigma)\rangle \rightarrow false}{\langle b_0 \textbf{ or } b_1, (sid, \sigma)\rangle \rightarrow false} \tag{5.43}$$

$$\frac{\langle b, (sid, \sigma)\rangle \rightarrow false}{\langle \textbf{ not } b, (sid, \sigma)\rangle \rightarrow true} \tag{5.44}$$

$$\frac{\langle b, (sid, \sigma)\rangle \rightarrow true}{\langle \textbf{ not } b, (sid, \sigma)\rangle \rightarrow false} \tag{5.45}$$

Table 5.9: *Boolean-related Semantic Rules.*

$$\begin{array}{c}
\langle f_0, (Messenger, \sigma'')\rangle \rightarrow true \qquad \langle f_0, (Messenger, \sigma'')\rangle \rightarrow true \\ \hline
\langle f_0 \textbf{ and } f_1, (Messenger, \sigma'')\rangle \rightarrow true \\ \hline
\langle b, (Messenger, \sigma'')\rangle \rightarrow true \\ \hline
\langle Send.click, (Messenger, \sigma'')\rangle \rightarrow true \qquad \langle b, (Messenger, \sigma'')\rangle \rightarrow true \\
validScr(MsgStatus) = true \qquad Messenger \neq MsgStatus \\ \hline
\langle tr_0, (Messenger, \sigma'')\rangle \rightarrow (MsgStatus, \sigma') \qquad compr(\sigma, \sigma') \\ \hline
\langle tr_0 tr_1, (Messenger, \sigma'')\rangle \rightarrow (MsgStatus, \sigma') \\ \hline
\langle w, (Messenger, \sigma)\rangle \rightarrow (Messenger, \sigma'') \\
\langle tr_0 tr_1, (Messenger, \sigma'')\rangle \rightarrow (MsgStatus, \sigma') \\ \hline
\langle \textbf{screen } Messenger\, u\; w\; tr_0 tr_1, (Messenger, \sigma)\rangle \rightarrow (MsgStatus, \sigma') \\ \hline
\langle s_0 s_1, (Messenger, \sigma)\rangle \rightarrow (MsgStatus, \sigma') \\ \hline
stop(appId) = false \qquad \langle s_0 s_1, (Messenger, \sigma)\rangle \rightarrow (MsgStatus, \sigma') \\ \hline
\langle appId\; s_0 s_1\;\; r, (Messenger, \sigma)\rangle \rightarrow \langle appId\; s_0 s_1\;\; r, (MsgStatus, \sigma')\rangle
\end{array}$$

Table 5.10: *Inference Tree of a snippet of the storyboard in Figure 5.4 interpreted with the semantic rules.*

when the app is stopped by the user or Android. In this configuration no other transition is possible and the app is said to be terminated. *Since no rule defined above, except rule 5.2, makes an app reach the terminal configuration, I say that the semantics ensure safety by construction.*

The semantic rules defined above ensure that an app is never "stuck", that is, if an app is in a valid configuration or a non-terminal configuration, then the app will either terminate or will be evaluated further in some configuration. I call this property of not getting stuck as progress. Formally, progress is specified as the following theorem:

Theorem 5.3.1. *If an app is in a configuration (sid, σ), then $\exists (sid', \sigma') : \langle app,\ (sid, \sigma)\rangle \rightarrow \langle app, (sid', \sigma')\rangle$, or $\langle app,\ (sid, \sigma)\rangle \rightarrow (\bot, erase(\sigma, \sigma))$*

I need the following lemma to prove theorem 5.3.1.

Lemma 5.3.2. *If an app is in a configuration (sid, σ) and $\exists tr \in TR : outTr(sid) = tr$, then $\exists (sid', \sigma\prime) : \langle tr, (sid, \sigma)\rangle \rightarrow (sid', \sigma')$, where $(sid', \sigma\prime)$ may or may not equal (sid, σ).*

Proof. I prove Lemma 5.3.2 by induction on the structure of tr.

Base Case: When tr is one transition from the screen sid, Lemma 5.3.2 is trivially true since either Rule 5.15, 5.16, 5.17, 5.18, or 5.19 will always apply.

Inductive Case: When tr is an ordered set of transitions, $tr_0 tr_1$, from the screen sid, then either apply rule 5.13 and 5.14 with the induction hypothesis on tr_0 or apply 5.14 with the induction hypothesis on tr_1.

Hence, by the principle of induction, Lemma 5.3.2 is true.

□

Proof. Let us assume that an app is in (sid, σ), where sid is a screen in the app and σ is the state of the app at screen sid.

Assume that there is an ordered set of transitions tr from screen sid, then by lemma 5.3.2, $\exists(sid', \sigma')$ such that the app will be further evaluated in (sid', σ') as per rule 5.3. If the app is stopped, then the app will terminate as per rule 5.2. □

5.3.3 Security Properties

An Android app often interacts with other apps on the device, the underlying platform, and remote servers. Such interaction involves sharing of information, responding to events, and performing tasks based on user actions. Many of these interactions have security implications that should be considered during app development. For example, can the user's personal information be stored safely on external storage? (information leak and data injection), who should have access to the content provided by the app? (permissions and privilege escalation), how should an app contact the server? (encryption).

While security implications are relevant, SeMA focuses on implications related to confidentiality and integrity of data.

Confidentiality *An app violates confidentiality if it releases data to an untrusted sink.* Hence, an app (and, consequently, its storyboard) that violates confidentiality is deemed as insecure. I explain the concepts of untrusted sinks in a storyboard in Section 5.3.4.

Integrity *An app violates integrity if it uses a value from an untrusted source.* Hence, an app (and, consequently, its storyboard) that violates integrity is deemed as insecure. I explain how a source is identified as untrusted in Section 5.3.4.

5.3.4 Analysis

There are multiple ways to check if extended storyboards satisfy various properties. In the current realization of SeMA, I use *information flow analysis* and *rule checking* to check and help ensure extended storyboards satisfy security properties concerning confidentiality and integrity.

Information Flow Analysis

This analysis tracks the flow of *information* in the form of values from sources to sinks in a storyboard.

A *source* is either a widget in a screen, an input parameter of a screen, or (the return value of) an operation. The set of sources in a storyboard are partitioned trusted and untrusted sources based on the guarantee of data integrity. Specifically, a source is *untrusted* if it is an input parameter of a screen's URI or it is an operation that reads data from an HTTP server, an open socket, device's external storage, or device's clipboard; or uses the capability of a resource provided by another app. All other sources are deemed as trusted.

A *sink* is either a widget in a screen or an argument to a screen or an operation. The set of sinks in a storyboard are partitioned trusted and untrusted sinks based on the guarantee of data confidentiality. Specifically, a sink is deemed as *untrusted* if it is an argument to an external screen identified without the *app* attribute or to an operation that writes data to an HTTP server, an open socket, device's external storage, or device's clipboard; or uses the capability of a resource provided by another app. All other sinks are deemed as trusted.

To reason about the flow of information between sources and sinks, I define a binary reflexive relation named *influences* between sources and sinks as follows: *influences(x,y)* (i.e., source x influences sink y) if

1. x is assigned to an input parameter y of screen s on an incoming transition to s,

2. x is an argument to operation y, or

3. value of operation x or input parameter x of screen s is assigned to widget y in screen s.

Here are few instances of this relation in Figure 5.4. *influences(y, Phone)* because input parameter y of *Contacts* screen is assigned to *Phone* widget. *influences(x,dispMsg)* because x is provided as an argument to operation *dispMsg*. *influences(dispMsg, Status)* because the value of operation *dispMsg* is assigned to *Status* widget in *SaveStatus* screen.

Further, all data flows inside the app are guaranteed to preserve the confidentiality and integrity of used/processed data. The operations provided as part of the capabilities of custom resources provided by the app are assumed to preserve the confidentiality and integrity of used/processed data.

With the above (direct) influence relation, guarantees, and assumptions, I use the transitive closure of *influences* relation to detect violation of security properties. I detect *potential violation of integrity* by identifying transitive (indirect) influences $(x, y) \in \textit{influences}^*$ in which x is an *untrusted* source. Likewise, I detect *potential violation of confidentiality* by identifying transitive (indirect) influences $(x, y) \in \textit{influences}^*$ in which y is an *untrusted* sink. All such identified indirect influences are reported to the developer and must be eliminated. Such an indirect influence can be eliminated by either replacing untrusted sources/sinks with trusted sources/sinks or indicating the indirect influence as safe by marking one or more of the direct influences that contribute the indirect influence as *safe*. When multiple sequences of direct influences between a source and a sink support an indirect influence, at least one direct influence in each sequence should be marked as safe to indicate the indirect influence is safe. This marking is similar to data declassification in traditional information flow analysis.

For example, in Figure 5.4, the y parameter of *Contacts* screen is untrusted as it is provided by an external app. So, the analysis will flag $\textit{influences}^*(y, \textit{Phone})$ as violating integrity due to the assignment of y to *Phone* widget in *Contacts* screen. A developer can fix this violation either by removing the assignment of y to *Phone* or marking the assignment as *safe* (as done in Figure 5.4).

Correctness argument The purpose of the analysis is to identify violations of confidentiality or integrity. This is done in two steps: 1) calculate the potential flows in the storyboard using the transitive closure of *influences* relation and 2) check if a flow involves an untrusted source or sink and is not marked as *safe*. Since the second step is based on pre-defined classification of sources and sinks and developer-provided *safe* annotations, the correctness of the analysis hinges on the first step.

If the *influences* relation correctly captures the direct flow between sources and sinks in a storyboard, then the transitive closure *influences** will capture all possible flows between the sources and sinks, including all flows violating confidentiality or integrity. Hence, *the analysis is complete* in identifying every violation of confidentiality or integrity.

The *influences** relation does not consider the effect of constraints on flows between sources and sinks (i.e., all constraints on transitions are assumed to be true). Consequently, the analysis may identify a flow between a source and a sink when there is no flow between the source and the sink (at runtime). For example, suppose a screen s has two incoming transitions i_1 and i_2 and two outgoing transitions o_1 and o_2 along with transition constraints that dictate transition o_x must be taken if and only if s was reached via transition i_x. Further, suppose an input parameter m of s is assigned a value along i_1 and i_2 and used as an argument along o_1 and o_2. In this case, the value assigned to m along i_1 (i_2) will not flow out of m along o_2 (o_1). However, the analysis will incorrectly identify the value assigned to m along i_1 (i_2) may flow out of m along o_2 (o_1). Since the analysis may report invalid violations, *the analysis is unsound.*

Formal Specification of the Analysis

This analysis tracks the flow of information in the form of values from *sources* to *sinks* in a storyboard, as defined in Section 5.3.4. In this section, I formally describe the algorithm.

Meta-language: Before specifying the algorithm, I define the following functions and structures required to understand it and the corresponding proof of correctness.

1. $Idn : Exp \rightarrow IDN$, where $Exp = W \cup P \cup F$ and $IDN = WID \cup VID \cup FN$, is a

function is used to obtain the identifier of a widget, input parameter, or an operation. It is defined as follows:

(a) *Widget* : if $w \in Exp$ and $w ::= wt\, wid\, x$, then $Idn(w) = gen(wid, \phi(wid))$

(b) *Parameter* : if $p \in Exp$ and $p ::= \textbf{param}\, vid\, x$, then $Idn(p) = gen(vid, \phi(vid))$

(c) *Operation* : if $f \in Exp$ and $f ::= \textbf{fun}\, fn\, rn.cn\, a$, then $Idn(f) = fn$

2. $FV : Exp \to 2^{IDN}$, is a function that maps $e \in Exp$ to the power set of IDN. FV is used to retrieve the identifiers that are used as values of a widget, input parameter, or operation. It is defined as follows:

   (a) if $e \in F$, then

      i. if $e ::= \textbf{fun}\, fn\, rn.cn\, vid$, then $FV(e) = \{gen(vid, \phi(vid))\}$

      ii. if $e ::= \textbf{fun}\, fn\, rn.cn\, wid$, then $FV(e) = \{gen(wid, \phi(wid))\}$

      iii. if $e ::= \textbf{fun}\, fn\, rn.cn\, f\prime$, then $FV(e) = \{Idn(f\prime)\}$

      iv. if $e ::= \textbf{fun}\, fn\, rn.cn\, a_0 a_1$, then $FV(e) = FV(f_0) \cup FV(f_0)$, where $f_0 ::= \textbf{fun}\, fn\, rn.cn\, a_0$ and $f_1 ::= \textbf{fun}\, fn\, rn.cn\, a_1$

      v. otherwise, $FV(e) = \emptyset$

   (b) if $e \in W$, then

      i. if $e ::= wt\, wid\, vid$, then $FV(e) = \{gen(vid, \phi(vid))\}$

      ii. if $e ::= wt\, wid\, f$, then $FV(e) = \{Idn(f)\}$

      iii. otherwise, $FV(e) = \emptyset$

   (c) if $e \in P$, then

      i. if $e ::= \textbf{param}\, vid_0\, vid_1$, then $FV(e) = \{gen(vid_1, \phi(vid_1))\}$

      ii. if $e ::= \textbf{param}\, vid\, wid$, then $FV(e) = \{gen(wid, \phi(wid))\}$

      iii. if $e ::= \textbf{param}\, vid\, f$, then $FV(e) = \{Idn(f)\}$

      iv. otherwise, $FV(e) = \emptyset$

3. The *safe* relation is used to collect direct flows between sources and sinks in the storyboard that have been marked by a developer with the *safe* keyword. Section 5.3.2 shows the syntax for marking direct flows as *safe*.

4. *safeParams* denotes the set of input parameters of a proxy screen $ps \in PS$ such that ps has been marked *safe* or ps has the *app* attribute set.

5. $CV : F \rightarrow BV$ is a predicate such that $CV(f) = \mathit{true}$ if operation f is an untrusted sink; $CV(f) = false$ otherwise.

6. $IV : F \rightarrow BV$ is a predicate such that $IV(f) = \mathit{true}$ if operation f is an untrusted source; $IV(f) = false$ otherwise.

Algorithm: The analysis algorithm proceeds in two stages. First, it captures the *direct* relationships between the identifier of a widget, an input parameter, or an operation with the identifier used as value in a widget, input parameter, or an argument to an operation in a binary relation called $influences$. It then uses *influences* to build a reflexive transitive closure $influences^*$ to capture the *indirect* relationships between widgets, input parameters, and operations. Second, for each widget, operation, and input parameter expression, the analysis collects their identifiers if they are directly or indirectly influenced by unsafe and untrusted identifiers. A violation is detected if a widget, input parameter, or operation has a non-empty set of such identifiers associated with it.

The algorithm is formally defined in the following steps:

1. Calculate the binary relation *influences*: $IDN \rightarrow IDN$ for each $e \in W \cup P \cup F$ as follows:

   (a) *Widgets* $w \in W$:

      i. if $w ::= \mathit{wt}\ \mathit{wid}\ \mathit{vid}$, then $\{(\mathit{gen}(\mathit{vid}, \phi(\mathit{vid})), \mathit{gen}(\mathit{wid}, \phi(\mathit{wid})))\}$.

      ii. if $w ::= \mathit{wt}\ \mathit{wid}\ f$, then $\{(Idn(f), \mathit{gen}(\mathit{wid}, \phi(\mathit{wid})))\}$.

      iii. if $w ::= w_0 w_1$, then $\mathit{influences}_{w0} \cup \mathit{influences}_{w1} \cup \mathit{influences}$

(b) *screen input parameters* $p \in P$:

i. if $p ::= \textbf{param}\, vid_0\, vid_1$, then $\{(gen(vid_1, \phi(vid_1)), gen(vid_0, \phi(vid_0)))\}$

ii. if $p ::= \textbf{param}\, vid\, wid$, then $\{(gen(wid, \phi(wid)), gen(vid, \phi(vid)))\}$

iii. if $p ::= \textbf{param}\, vid\, f$, then $\{(Idn(f), gen(vid, \phi(vid)))\}$

iv. if $p ::= p_0p_1$, then $\mathit{influences}_{p0} \cup \mathit{influences}_{p1} \cup \mathit{influences}$

(c) *Operations* $f \in F$:

i. if $f ::= \textbf{fun}\, fn\; rn.cn\; vid$, then $\{(gen(vid, \phi(vid)), fn)\}$

ii. if $f ::= \textbf{fun}\, fn\; rn.cn\; wid$, then $\{(gen(wid, \phi(wid)), fn)\}$.

iii. if $f ::= \textbf{fun}\, fn\; rn.cn\; f\prime$, then $\{(Idn(f\prime), fn)\}$

iv. if $f ::= \textbf{fun}\, fn\; rn.cn\; a_0a_1$, then $\mathit{influences}_{f0} \cup \mathit{influences}_{f1} \cup \mathit{influences}$, where $f_0 ::= \textbf{fun}\, fn\; rn.cn\; a_0$ and $f_1 ::= \textbf{fun}\, fn\; rn.cn\; a_1$

2. Calculate $\mathit{influences}^*$, the reflexive transitive closure of the binary relation *influences*.

3. Mark untrusted sources using the $\mathit{Untrusted} : IDN \rightarrow BV$ predicate for each $e \in F \cup U$ as follows:

(a) if $e \in F \wedge IV(e)$, then $\forall(Idn(f), k) \in \mathit{influences}^* : \mathit{Untrusted}(k) = \mathit{true}$

(b) if $e ::= us/K$, then $\forall k \in K : \mathit{Untrusted}(k) = \mathit{true} \wedge \forall(gen(vid, \phi(vid), y) \in \mathit{influences}^* : \mathit{Untrusted}(y) = \mathit{true}$

4. Collect flows marked *safe* for each $e \in W \cup F \cup P$ as follows:

(a) if $e ::= \mathit{safe}\, wt\, wid\, x$, then $\{x, gen(wid, \phi(wid)))\} \cup \mathit{safe}$ and $\forall(gen(wid, \phi(wid)), k) \in \mathit{influences}^* : \{(gen(wid, \phi(wid)), k)\} \cup \mathit{safe}$

(b) if $e ::= w_0w_1$, then $\mathit{safe}_{w0} \cup \mathit{safe}_{w1} \cup safe$

(c) if $e ::= \textbf{fun}\, fn\; rn.cn\; \mathit{safe}\; x$, then $\{(gen(x, \phi(x)), fn)\} \cup \mathit{safe}$, where $x \in (WID \cup VID)$

(d) if $e ::= \textbf{fun}\, fn\; rn.cn\; \mathit{safe}\; f$, then $\{(Idn(f), fn)\} \cup \mathit{safe}$

(e) if $e ::= \textbf{fun}\, \mathit{fn}\, \mathit{rn.cn}\, a_0 a_1$, then $\mathit{safe}_{e0} \cup \mathit{safe}_{e1} \cup \mathit{safe}$, where $e0 ::= \textbf{fun}\, \mathit{fn}\, \mathit{rn.cn}\, a_0$ and $e1 ::= \textbf{fun}\, \mathit{fn}\, \mathit{rn.cn}\, a_1$

(f) if $e ::= \textbf{param}\, \mathit{vid}\, x$ and $\mathit{gen}(\mathit{vid}, \phi(\mathit{vid})) \in \mathit{safeParams}$, then $\{(x, \mathit{gen}(\mathit{vid}, \phi(\mathit{vid}))) \in \mathit{safe}\}$

(g) if $e ::= p_0 p_1$, then $\mathit{safe}_{p0} \cup \mathit{safe}_{p1} \cup \mathit{safe}$

5. Calculate the function $\mathit{IF} : \mathit{Exp} \to 2^{IDN}$ for each $e \in W \cup F \cup P$ as follows:

(a) if $e \in W$, then

i. if $e ::= \mathit{wt}\, \mathit{wid}\, \mathit{vid}$, then
$\mathit{IF}(e) = \{\mathit{gen}(\mathit{wid}, \phi(\mathit{wid})) \mid \mathit{Untrusted}(\mathit{vid}) \wedge (((\mathit{vid}, \mathit{wid}) \notin \mathit{safe}) \wedge (\exists (z, \mathit{vid}) : (z, \mathit{vid}) \in \mathit{influences} \implies (\forall (y, z) : (y, z) \in \mathit{influences}^* \implies (y, z) \notin \mathit{safe})))$

ii. if $e ::= \mathit{wt}\, \mathit{wid}\, f$, then
$\mathit{IF}(e) = \{\mathit{gen}(\mathit{wid}, \phi(\mathit{wid})) \mid IV(f) \wedge (Idn(f), \mathit{wid}) \notin \mathit{safe}$

iii. if $e ::= w_0 w_1$, then $\mathit{IF}(e) = \mathit{IF}(w_0) \cup \mathit{IF}(w_1)$

(b) if $e \in F$, then

i. if $e ::= \textbf{fun}\, \mathit{fn}\, \mathit{rn.cn}\, \mathit{wid}$, then
$\mathit{IF}(e) = \{\mathit{fn} | (\mathit{wid}, \mathit{fn}) \notin \mathit{safe} \wedge Idn(w) = \mathit{wid} \wedge \mathit{IF}(w) \neq \emptyset\}$
$\cup \{\mathit{fn} | CV(e) \wedge ((\mathit{wid}, \mathit{fn}) \notin \mathit{safe}) \wedge (\exists (\mathit{fn}', \mathit{wid}) : (\mathit{fn}', \mathit{wid}) \in \mathit{influences} \implies (\mathit{fn}', \mathit{wid}) \notin \mathit{safe}) \wedge (\exists (y, \mathit{vid}), (\mathit{vid}, \mathit{wid}) : ((y, \mathit{vid}) \in \mathit{influences} \wedge (\mathit{vid}, \mathit{wid}) \in \mathit{influences}) \implies (\forall (k, \mathit{vid}) : (k, \mathit{vid}) \in \mathit{influences}^* \implies (k, \mathit{vid}) \notin \mathit{safe}))\}$

ii. if $e ::= \textbf{fun}\, \mathit{fn}\, \mathit{rn.cn}\, \mathit{vid}$, then
$\mathit{IF}(e) = \{\mathit{fn} | (\mathit{Untrusted}(\mathit{vid}) \vee CV(e)) \wedge (\mathit{vid}, \mathit{fn}) \notin \mathit{safe} \wedge (\exists (z, \mathit{vid}) : (z, \mathit{vid}) \in \mathit{influences} \implies (\forall (y, z) : (y, z) \in \mathit{influences}^* \implies (y, z) \notin \mathit{safe}))\}$

iii. if $e ::= \textbf{fun}\, \mathit{fn}\, \mathit{rn.cn}\, f$, then
$\mathit{IF}(e) = \{\mathit{fn} | (IV(f) \vee CV(e)) \wedge (Idn(f), \mathit{fn}) \notin \mathit{safe}\}$

iv. if $e ::= \textbf{fun}\, \mathit{fn}\, \mathit{rn.cn}\, a_0 a_1$, then
$\mathit{IF}(e) = \mathit{IF}(e_0) \cup \mathit{IF}(e_1)$, where $e_0 ::= \textbf{fun}\, \mathit{fn}\, \mathit{rn.cn}\, a_0$ and $e_1 ::= \textbf{fun}\, \mathit{fn}\, \mathit{rn.cn}\, a_1$

(c) if $e \in P$, then

i. if e ::= **param** $vid\, id$, where $id \in (WID \cup VID)$, then $\mathit{IF}(e) = \{gen(vid, \phi(vid)) \mid (id, vid) \in \mathit{safe} \vee (\forall(y, id) \in \mathit{influences}^* : (y, id) \notin \mathit{safe})\}$

ii. if e ::= **param** $vid\, f$, then $\mathit{IF}(e) = \{gen(vid, \phi(vid)) \mid Idn(f) = fn \wedge (fn, vid) \in \mathit{safe}$

iii. if $e ::= e_0 e_1$, then $\mathit{IF}(e) = \mathit{IF}(e_0) \cup \mathit{IF}(e_1)$

The analysis reports a violation if $\exists e \in Exp : \mathit{IF}(e) \neq \emptyset$ in the storyboard.

As an example, consider information flow analysis of the storyboard in Figure 5.4. As per the algorithm, I first build the binary relation:

$$\mathit{influences} = \{(y, Phone), (Phone, z), (Phone, x), (dispMsg, Status), (x, dispMsg), \\ (Phone, savePhone), (getContacts, sendMsg)\}$$

Next, a reflexive transitive closure is calculated as follows:

$$\mathit{influences}^* = \{(y, Phone), (Phone, z), (Phone, x), (dispMsg, Status), (x, dispMsg), \\ (Phone, savePhone), getContacts, sendMsg), (y, z), (y, x), (Phone, dispMsg), \\ (y, dispMsg), (y, Status), (x, Status)), ...\}$$

The predicates IV and CV return *false* for every operation in the storyboard since none of them use resources that correspond to untrusted source/sink. Since, y is a variable from an external app, $\forall(y, k) \in \mathit{influences}^* : \mathit{Untrusted}(y, k) = \mathit{true}$.

Table 5.11 shows the result of computing IF for each $e \in Exp$ in the storyboard before any flow was marked as *safe*. Since $\exists e \in Exp : \mathit{IF}(e) \neq \emptyset$, the analysis reports a violation.

Exp	*IF*	*IF* (safe)
TextView Phone y	{gen(Phone,$\phi(Phone)$)}	{}
TextView Status "MsgSent"	{}	{}
TextView Status dispMsg(x)	{}	{}
sendMsg(...)	{}	{}
getContacts(...)	{}	{}
dispMsg(x)	{Idn(dispMsg)}	{}
param z Phone	{}	{}
param x Phone	{}	{}

Table 5.11: *Information Flow Analysis of Storyboard shown in Figure 5.4. The first column indicates an $e \in Exp$ (i.e., widget, operation, or input parameter of a screen). The second column indicates a set obtained by evaluating IF(e) assuming that nothing was marked as safe. The third column indicates IF with a flow marked as safe.*

To fix the violation, the variable y needs to be obtained from a trusted source or the flow/s related to y should be marked *safe*. If we take the latter approach and mark the flow between y and the widget *Phone* as *safe* as shown in Figure 5.4, then on running the analysis again, *safe* is as follows:

$$safe = \{(y, Phone), (Phone, z), (Phone, x), ...\}$$

Since $\forall e \in Exp : IF(e) = \emptyset$ as shown in Table 5.11 third column, no violations are reported by the analysis.

Proof of Correctness

The correctness of the function *IF* hinges on the calculation of the transitive closure *influences**, predicate *Untrusted*, and set *safe*. Since *Untrusted* and *safe* are based on a pre-defined list of trusted and untrusted identifiers and developer-provided annotations/indicators, it is enough to prove that the relations captured in *influences** reflect the flow as specified in the semantics.

Theorem 5.3.3. $\forall a \in APP, \forall sid, sid' \in SID, x, y \in ID, \sigma, \sigma' : \langle (a),\ (sid, \sigma)\rangle \rightarrow^* \langle a, (sid', \sigma')\rangle \wedge \sigma(x) = \sigma'(y) \implies (x, y) \in influences^*$

Proof. I prove this by induction on the no. of steps it takes an app from the current configuration (sid, σ) to reach a configuration (sid', σ').

Base Case: Assume that a is an app, $sid, sid' \in SID, x, y \in ID, \sigma, \sigma'$ such that $\langle a, \ (sid, \sigma)\rangle \rightarrow \langle a, (sid', \sigma')\rangle \wedge \sigma(x) = \sigma'(y)$. In screen sid', $\exists l$ such that l is a widget or provides an argument to an input parameter of another screen as part of a transition from sid'. Since $\sigma(x) = \sigma'(y)$, $y = Idn(l)$ and $k \in FV(l)$, where either $k = x$ or k is the identifier of an operation with an argument x. By the definition of *influences*, either $(x, y) \in$ *influences* or $(x, k) and (k, y) \in$ *influences*. In either case, $(x, y) \in$ *influences**.

Therefore, 5.3.3 is true.

Inductive Case:

- Case 1: Let's assume that a is an app, $sid, sid' \in SID, x, z \in ID, \sigma, \sigma''$ such that $\langle (a), \ (sid, \sigma)\rangle \rightarrow^k \langle a, (sid', \sigma'')\rangle \wedge \sigma(x) = \sigma''(z) \implies (x, z) \in$ *influences**, where k is the no. of steps it takes to reach from σ to σ''. This is our induction hypothesis.

  Further, assume that $\exists sid'' \in SID, y \in ID, \sigma'$ such that $\langle a, \ (sid', \sigma'')\rangle \rightarrow \langle a, (sid'', \sigma') \wedge \sigma''(z) = \sigma'(y)$ and l is a widget in screen sid'' or provides an argument to the input parameter of a screen via a transition from sid''. Since $\sigma''(z) = \sigma'(y)$, $y = Idn(l)$ and $k \in FV(l)$, where $k = z$ or k is the identifier of an operation that uses z as an argument. In either case, $(z, y) \in$ *influences* due to the definition of *influences*. Since *influences* $\subseteq$ *influence**, $(z, y) \in$ *influences**. From the induction hypothesis, $(x, z) \in$ *influences**. Since, *influences** is a transitive closure, $(x, y) \in$ *influences**.

- Case 2: Let's assume that a is an app, $sid, sid' \in SID, \sigma, \sigma'$ such that $\langle (a), \ (sid, \sigma)\rangle \rightarrow^m \langle a, (sid'\rangle, \sigma') \wedge \sigma(x) = \sigma'(z) \implies (x, z) \in$ *influences**, and $sid'' \in SID, \sigma''$ such that $\langle (a), \ (sid', \sigma')\rangle \rightarrow^n \langle a, (sid'', \sigma'')\rangle \wedge \sigma'(z) = \sigma''(j) \implies (z, j) \in$ *influences**, and $k = m + n$.

  Further, assume that $\exists sid''' \in SID, y \in ID, \sigma''' : \langle a, \ (sid'', \sigma'')\rangle \rightarrow \langle a, (sid''', \sigma''') \wedge \sigma''(j) = \sigma'''(y)$ and l is a widget in screen sid''' or provides the argument to an input parameter of a screen via a transition from sid'''. Since $\sigma''(j) = \sigma'''(y)$, $y = Idn(l)$ and

$k \in FV(l)$, where $k = j$ or k is the identifier of an operation that uses j as an argument. In either case, $(j, y) \in$ *influences* by the definition of *influences*. Since *influences* $\subseteq$ *influence**, $(j, y) \in$ *influences**. From the induction hypothesis, $(x, z) \in$ *influences** and $(z, j) \in$ *influences**. Since, *influences** is a transitive closure, $(x, y) \in$ *influences**.

Conclusion: By the principle of induction 5.3.3 is true.

□

The converse of Theorem 5.3.3 does not hold for the analysis because the analysis does not consider the effects of constraints when building the *influences* relation. So, it is possible that an ID x *influences* y even if the semantics does not allow x to flow into y. For example, let us assume that a transition t is guarded by a constraint b and when b is *true*, an argument a is passed to the destination screen. Let's also assume that $x \in FV(a)$ and $y = Idn(a)$. Finally, let us assume that b is always *false*. In such a scenario, x *influences* y but the semantics will not allow x to flow into y. This implies that the analysis will flag violations even if there isn't any. However, the developer can override the violation by setting the *safe* attribute appropriately.

Rule Checking

Prior research has developed guidelines and best practices for secure Android app development [39, 40]. Based on these standards, I have developed rules that can be enforced at design time to prevent the violation of properties related to confidentiality and integrity.

Following is the list of rules supported by the current realization of SeMA along with the reasons for the rules.

1. *Capabilities offered by custom resources must be protected by access control.* If any external client can access a custom resource (i.e., its *access* attribute is set to *all*) and the resource offers privileged capabilities, then malicious clients can gain access to privileged capabilities without the user's consent. Further, *Android's policy of least privilege* stipulates that apps should have minimal privileges and acquire the privileges required to use protected services.

2. *WebView widgets must be configured with a whitelist of URL patterns.* A WebView widget in an app works like a browser – it accepts a URL and loads its contents – but it does not have many of the security features of full-blown browsers. Also, a WebView widget has the same privileges as the containing app, has access to the app's resources, can be configured to execute JavaScript code. Hence, loading content from untrusted sources into WebView widgets facilitates exploitation by malicious content.

3. *Operations configured to use HTTPS remote servers must use certificate pinning.* HTTPS remote servers are signed with digital certificates issued by certificate authorities (CAs). Android defines a list of trusted CAs and verifies that the certificate of an HTTPS remote server is signed with a signature from a trusted CA. However, if a trusted CA is compromised, then it can be used to issue certificates for malicious servers. Hence, to protect against such situations, certificates of trusted servers are pinned (stored) in apps and only servers with these certificates are recognized as legit servers by the apps.

4. *Operations configured to use SSL sockets must use certificate pinning.* The reasons from the case of certificate pinning for HTTPS applies here as well.

5. *Cipher operations must use keys stored in secure key stores (containers).* The results of cipher operations can be influenced by tampering the cryptographic keys used in cipher operations. Further, since cryptographic keys are often used across multiple executions of an app, they need to be stored in secondary storage that is often accessible by all apps on a device. Hence, to protect against unwanted influences via key tampering, cipher keys should be stored in secure key stores (containers).

Realization of Rule Checking Violations of rule 1 are detected by checking if a custom resource offers a privileged capability and has its *access* attribute set to *all*.

The *trust-patterns* attribute of *WebView* widget is used to specify the whitelist of trusted URL patterns. Violations of rule 2 is detected by checking if *trust-patterns* attribute is specified for every *WebView* widget.

Violations of rule 5 are detected by checking if the key argument provided to a cipher operation is the value returned by a pre-defined operation to keys from a secure container.

Violations of rules 1, 2, and 5 are flagged as errors and must be addressed before moving to the code generation phase.

Certificate pinning is enabled by default in every storyboard in the methodology. However, since techniques other than certificate pinning can be used to secure connections to servers, a developer can disable certificate pinning by setting *disableCertPin* attribute in a network-related operation. Such cases are detected as violations of rules 3 and 4. They are flagged as warnings but do not inhibit the developer from moving to the code generation phase.

5.3.5 Code Generation

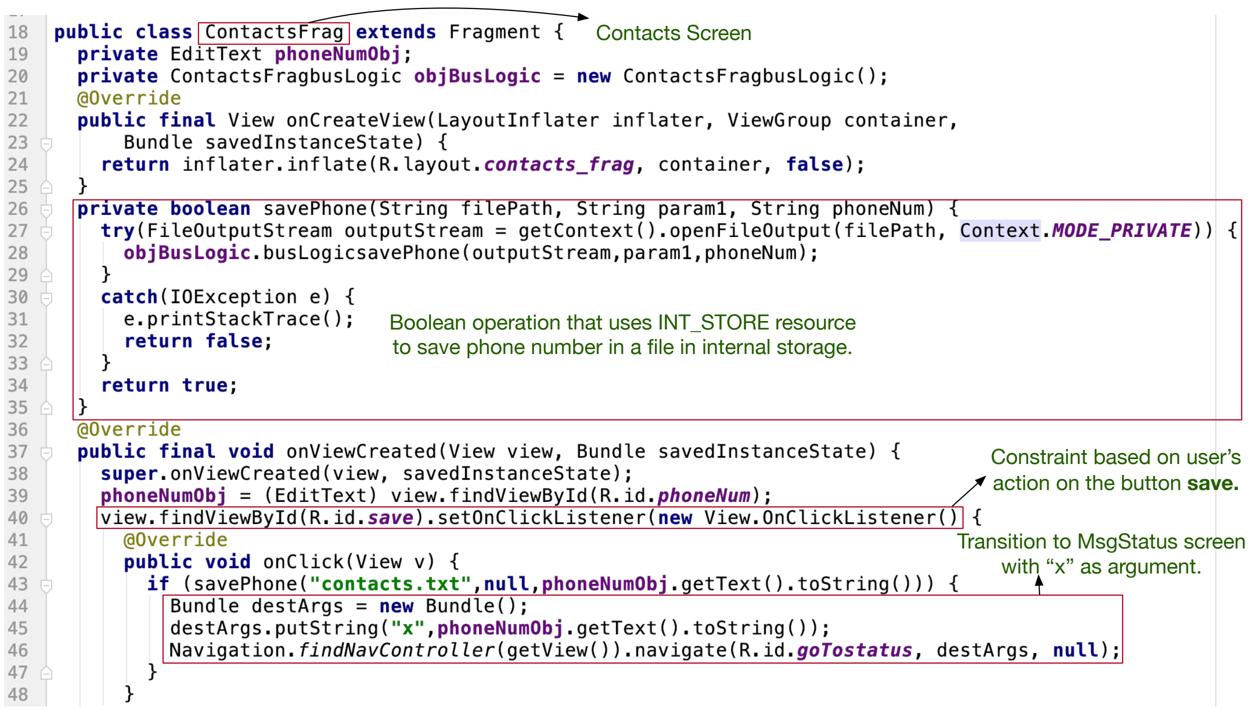
Once the developer has verified that the specified storyboard does not violate properties related to confidentiality and integrity, she can generate code from the storyboard. Figure 5.6 shows a fragment of generated code for the running example (Figure 5.4).

Mapping and Translation Rules

The current realization of SeMA hinges on various choices in mapping and translating storyboard-level entities and concepts into code-level entities and concepts. These choices are influenced by the semantics outlined in Section 5.3.2 and encoded in the following mapping and translation rules used during code generation.

1. A Screen is translated to a `Fragment`. For each input parameter of the screen, a function to obtain the value of the parameter is generated. If an input parameter is not available at runtime, then the corresponding function returns `null`.

2. A widget is translated to the corresponding widget type in Android (e.g., a widget displaying text is translated to `TextView`). The value of the widget is the corresponding value specified in the storyboard. For example, if the value is provided by a screen's



```
public class ContactsFrag extends Fragment {
  private EditText phoneNumObj;
  private ContactsFragbusLogic objBusLogic = new ContactsFragbusLogic();
  @Override
  public final View onCreateView(LayoutInflater inflater, ViewGroup container,
      Bundle savedInstanceState) {
    return inflater.inflate(R.layout.contacts_frag, container, false);
  }
  private boolean savePhone(String filePath, String param1, String phoneNum) {
    try(FileOutputStream outputStream = getContext().openFileOutput(filePath, Context.MODE_PRIVATE)) {
      objBusLogic.busLogicsavePhone(outputStream,param1,phoneNum);
    }
    catch(IOException e) {
      e.printStackTrace();
      return false;
    }
    return true;
  }
  @Override
  public final void onViewCreated(View view, Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);
    phoneNumObj = (EditText) view.findViewById(R.id.phoneNum);
    view.findViewById(R.id.save).setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
        if (savePhone("contacts.txt",null,phoneNumObj.getText().toString())) {
          Bundle destArgs = new Bundle();
          destArgs.putString("x",phoneNumObj.getText().toString());
          Navigation.findNavController(getView()).navigate(R.id.goTostatus, destArgs, null);
        }
      }
```

Contacts Screen
Boolean operation that uses INT_STORE resource to save phone number in a file in internal storage.
Constraint based on user's action on the button **save.**
Transition to MsgStatus screen with "x" as argument.

Figure 5.6: *Code generated for the Contacts screen in the storyboard depicted in Figure 5.4*

input parameter x, then the return value from the getter function of x is set as the widget's value (e.g., `TextView.setText(getX())`. The value in a widget is obtained via the corresponding getter function (e.g., `TextView.getText()`). If the corresponding value is `null`, then the widget will have a default value.[2]).

3. The constraint associated with a transition from a source screen to a destination screen is a conjunction of a user action and boolean operations. The user action part of the constraint is translated to a listener/handler function in the source screen that is triggered by the corresponding user action (e.g., button click). If the constraint has a boolean operation, then a conditional statement is generated with the boolean operation as the condition in the body of the listener function. The *then* block of the conditional statement has the statements required to trigger the destination screen. If the constraint has no boolean operations, then the body of the listener function has statements required to trigger the destination screen. If the constraint has no user action, then the checks corresponding to the boolean operations are performed when

[2]Every widget in Android has a default value (e.g., a `TextView` has empty string as default value.

the source fragment is loaded.

I use Android's navigation APIs to trigger a destination screen. If the destination screen is a proxy screen, then intents are used to trigger the destination screen determined by the *URI* and *app* attribute. Arguments to destination screens are provided as key/value pairs bundled via the `Bundle` API.

When a screen has multiple outgoing transitions, the statements corresponding to the transitions are chained in the specified order of the transitions in the storyboard.

4. An operation is translated to a function with appropriate input parameters and return value. Each reference to the operation in a storyboard is translated to call the corresponding function.

   The type of the input parameters depends on the type of the arguments provided to the function. For example, if the argument is provided by a widget that displays text, then the type of the parameter will be `String`. The return type depends on how the function is used. For example, if the function is used as a boolean operation in a constraint, then its return type will be `boolean`. If the function is assigned to a widget that displays text, then the function's return type will be `String`.

   If the operation uses a capability provided by a pre-defined resource, then the body of the corresponding function will contain the statements required to use the capability. Otherwise, the function will have an empty body that needs to be later filled in by the developer. For example, on line 27 in Figure 5.6, function *savePhone* contains the statement required to create a file in the device's internal storage since the same operation uses *write* capability of the resource *INT_STORE* in the storyboard.

   If the operation raises an exception, then the exception is caught and `null` is returned for a non-boolean operation and `false` is returned for a boolean operation.

5. A developer can extend the generated definitions of functions. For example, on line 28 in Figure 5.6, the generated code provides a hook for the developer to extend *savePhone*.

6. A custom resource is translated to an appropriate Android component. The capabilities provided by a custom resource can be accessed via an Android intent. Currently, I only support broadcast receivers as custom resources.

7. The use of a resource in the storyboard indicates an app depends on the resource. Such dependencies are captured in the app's configuration during code generation while relying on the Android system to satisfy these dependencies at runtime in accordance with the device's security policy (e.g., grant permission to use a resource at install time).

5.4 Canonical Examples

I illustrate SeMA with a couple of canonical examples. Each example demonstrates a vulnerability that can be prevented by the methodology at design time. Each example will have a description of the expected behavior, a step-wise explanation of how the expected behavior is specified in SeMA, and how SeMA helps uncover security violations in the specified behavior.

5.4.1 Data Injection Example

Consider an app that allows users to log in and view their profile information. From another app, a valid user of the app can navigate only to the screen showing profile information. The example is based on the fragment injection vulnerability discovered in real-world apps [120].

The app will be specified in SeMA as follows:

Screens and transitions: As the first step, a developer specifies the screens of the app, the widgets in each screens, and the possible transitions between the screens. As shown in Figure 5.7, the initial storyboard has 4 screens – `Start`, `LoginFrag`, `Home`, and `Profile`. Each screen has widgets (e.g., the `Start` screen has one button with label `Launch`). Finally, the screens are connected to each other via transitions to indicate how the user can navigate between screens. For example, from the `LoginFrag`, a user can navigate to either `Home` or `Profile`.

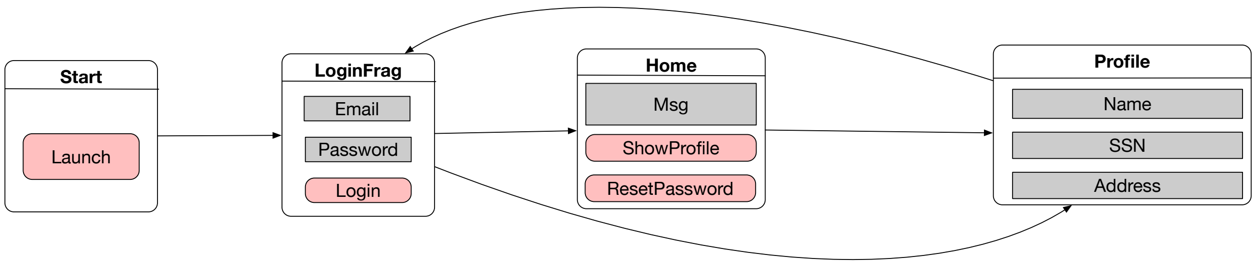


Figure 5.7: *Data Injection Example: Initial Storyboard Design*

Extend with user-controlled actions: A developer adds any constraints related to a user's action to the transitions. Such actions correspond to actions/gestures performed by users on the widgets in the screens. In Figure 5.8 such constraints are highlighted in orange. For example, the transition from `Start` to `LoginFrag` is taken when `Launch` button in `Start` is clicked.

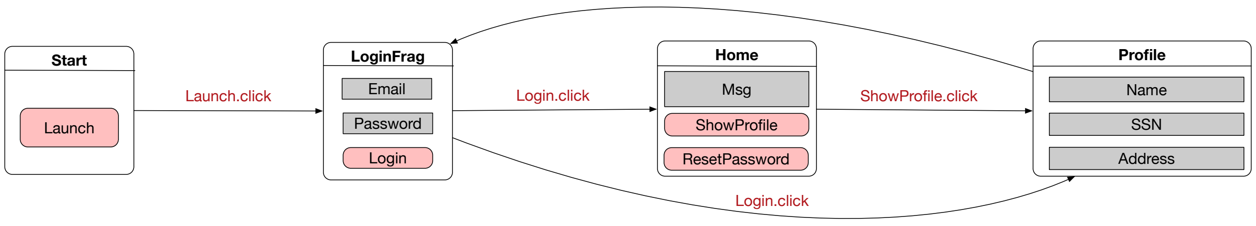


Figure 5.8: *Data Injection Example: Adding user-related constraints*

Add operations as constraints: A developer adorns transitions with operations that return boolean values. Figure 5.9 demonstrates these extensions (highlighted in blue). For example, the transitions from `LoginFrag` to `Home` is taken when the `Login` button is clicked, the `verify` operation, and the `isFragHome` operation returns true. Likewise, the transition from `Profile` to `LoginFrag` is taken when the `validToken` operation returns true. These boolean operations may have input parameters. For example, the `verify` operation takes two input parameters, the arguments to which are provided by the values in widgets `Email` and `Password`.

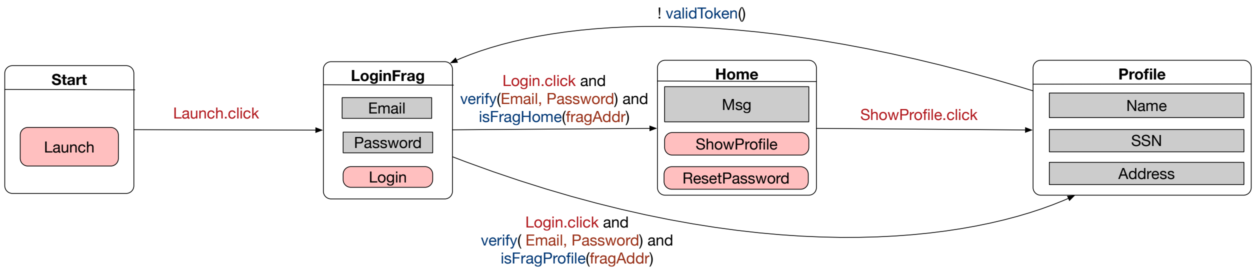
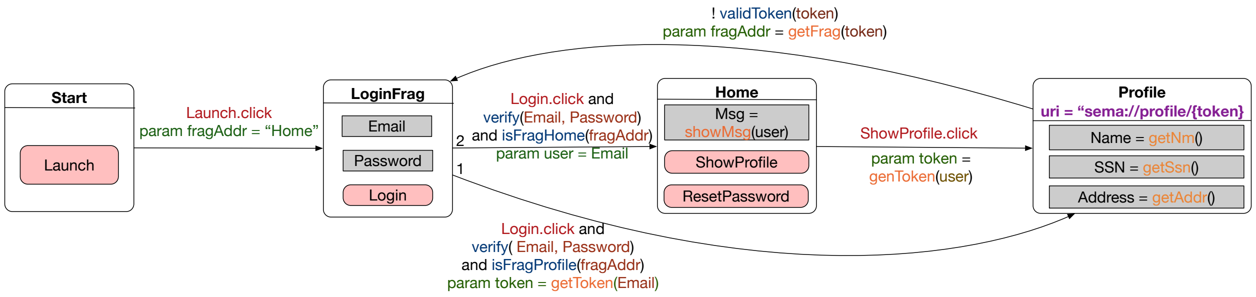


Figure 5.9: *Data Injection Example: Adding boolean constraints*

Connect screen input parameters to data sources: A developer specifies data sources as arguments to a screen's input parameters. Arguments are provided in two ways – (1) as part of incoming transitions to a screen, or (2) as part of URIs associated with screens. In Figure 5.10, arguments provided as part of transitions are highlighted in green and arguments associated with URIs are shown in purple. For example, the `token` input parameter of the screen `Home` is provided the value of a non-boolean operation, `getToken(Email)` (highlighted in orange), as argument associated with the transition from `LoginFrag` to `Profile`. Likewise, if the transition from `Home` to `Profile` is taken, then `token` is provided the value from `getToken(user)` as argument.



! validToken(token)
param fragAddr = getFrag(token)
Start
Launch
Launch.click
param fragAddr = "Home"
LoginFrag
Email
Password
Login
Login.click and verify(Email, Password) and isFragHome(fragAddr)
param user = Email
2
1
Home
Msg = showMsg(user)
ShowProfile
ResetPassword
ShowProfile.click
param token = genToken(user)
Profile
uri = "sema://profile/{token}
Name = getNm()
SSN = getSsn()
Address = getAddr()
Login.click and verify(Email, Password) and isFragProfile(fragAddr)
param token = getToken(Email)

Figure 5.10: *Data Injection Example: Input Parameters, URIs, and Non-boolean operations*

Analyze for security violations: A developer analyzes the specified behavior in the storyboard to verify the properties related to confidentiality and integrity. Figure 5.11 shows that there is a violation of integrity in the storyboard as `isFragProfile` and `isFragHome` operations consume the input parameter `fragAddr` as argument. On the transition from `Profile` screen to `LoginFrag` screen, `fragAddr` takes on the return value of `getFrag` operation that consumes `token` parameter of `Profile` screen. Since an external app provides the `token` argument when `Profile` screen triggered via its URI, an external app can manipulate `token` to gain access to `Home` screen. Information flow analysis will detect and flag this violation by following the chain of flow from untrusted source and sinks.

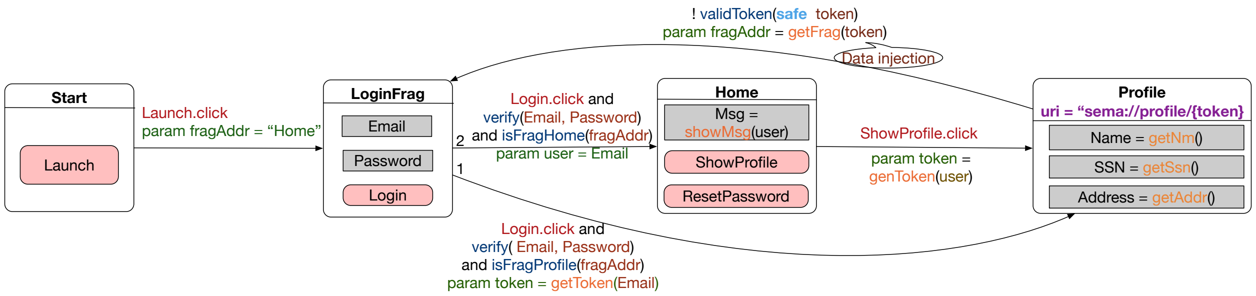


Figure 5.11: *Data Injection Example: Security Analysis*

Apply fix suggested by SeMA: This vulnerability can be fixed by changing `param fragAddr = getFrag(token)` to `param fragAddr = "profile"` on the transition from `Profile` screen to `LoginFrag` screen as this breaks the dependence between the app's navigation and `token` parameter.

Apply alternate fix: While the above fix will make the app more secure, SeMA will still flag a vulnerability because the transition from `Profile` to `LoginFrag` is not guarded by a user-related constraint. SeMA allows developers to override such warnings. However, it is advisable not to do so. In this context, a more secure design would be to have a separate screen that interacts with external apps. Figure 5.12 shows this design, which is more secure because an external app cannot navigate to private screens in the app. While an external app can influence the result of the boolean operation `validToken`, this is required since the app needs to verify the provided token. To allow this requirement, the developer can let SeMA know that the flow of `token` to `validToken` is `safe` (highlighted in blue in Figure 5.12).

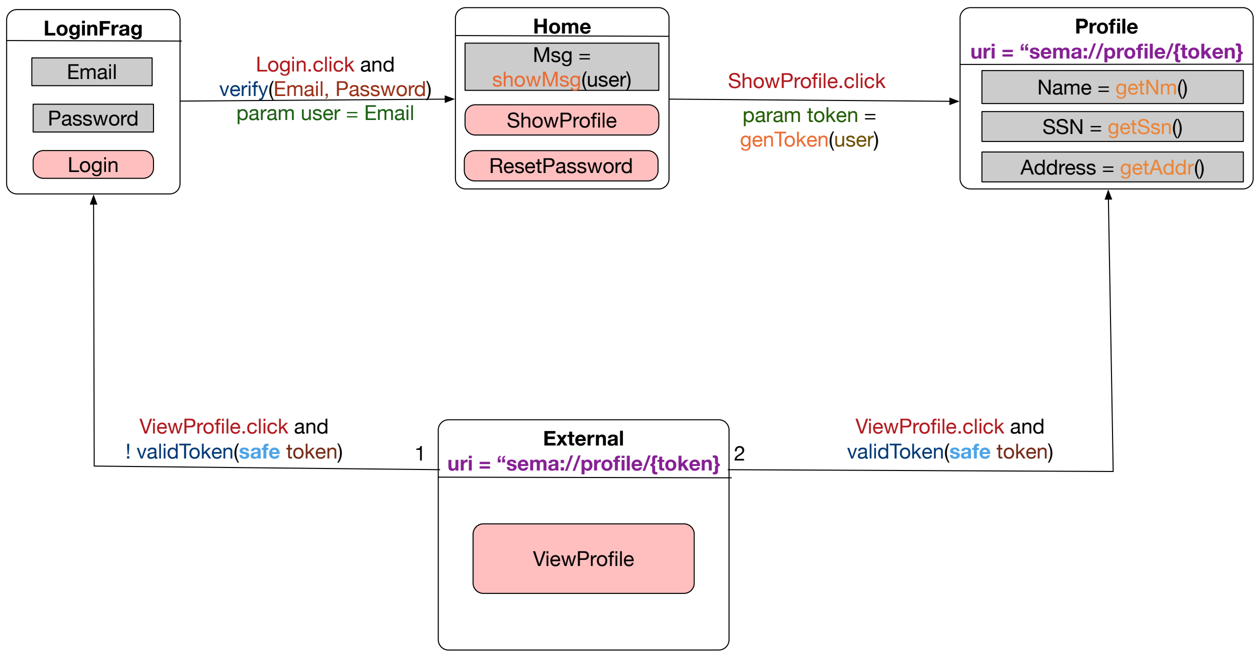


Figure 5.12: *Data Injection Example: Secure Version*

5.4.2 Data Leak Example

Consider an app that allows a user to enter a trusted URL in a text field and displays the content from the URL. If the URL is a file URL, then the data in the file is displayed to the user and downloaded to the device's external storage. The URL can be a file URL or a web URL. The example is based on the sensitive data leak vulnerability discovered in Firefox app for Android [121] and the Zomato app [122].

The app will be specified in SeMA as follows:

Screens and transitions: A developer starts by specifying the screens, widgets in each screens, and the possible transitions between the screens in an initial storyboard as shown on Figure 5.13. The app has 3 screens – `Home`, `Display`, and `DisplayFile`. The `Display` screens is used to display web content from a URL entered in `Url` in `Home` screen. The `DisplayFile` screen is used to display the contents of a file entered by the user in `Url` in `Home` screen.

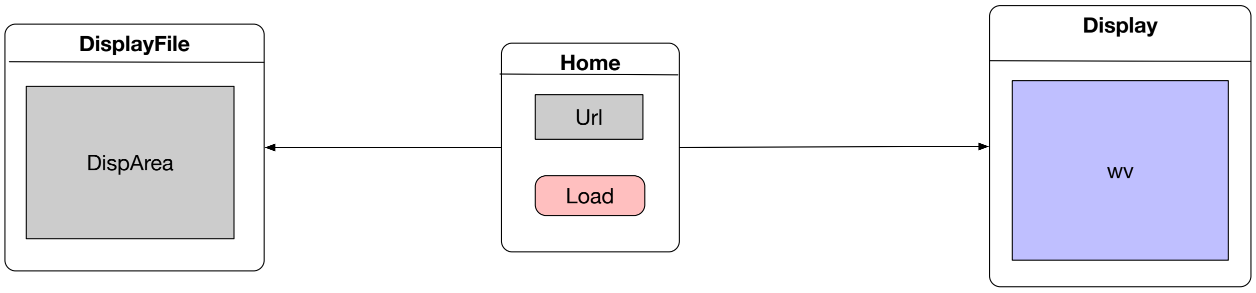


Figure 5.13: *Data Leak Example: Initial Storyboard*

Add user-controlled constraints: As described previously a developer adds user-controlled actions as constraints to transitions. In the figure, a user can either navigate to `Display` or `DisplayFile` from `Home` upon the click of the `Load` button in `Home`.

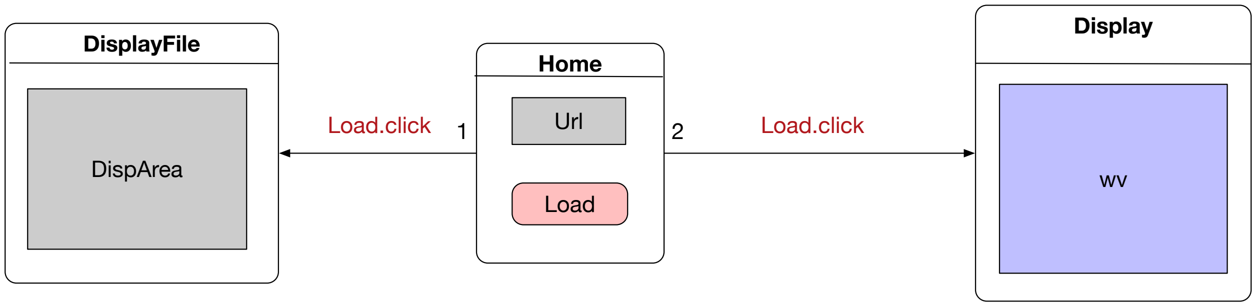


Figure 5.14: *Data Leak Example: Adding user-related constraints*

Add operations as constraints: Figure 5.9 shows these extensions (highlighted in blue). For example, the transitions from `Home` to `Display` is taken when the `Load` button is clicked and the `startsWith` operation returns true. The `startsWith` verifies if the URL entered by the user starts with "http". Likewise, the transition from `Home` to `DisplayFile` is taken when the `Load` button is clicked, the entered URL starts with "file" (see `startsWith` operation in Figure 5.15), and the `save` operation returns true (i.e., the content in the file path `Url` is saved).

Add resources to operations: A developer adds more detail to the specified operations via pre-defined resources. For example, a developer specifies the type of storage that will be used by the `save` operation as shown in Figure 5.16 (highlighted in purple). In the figure, the

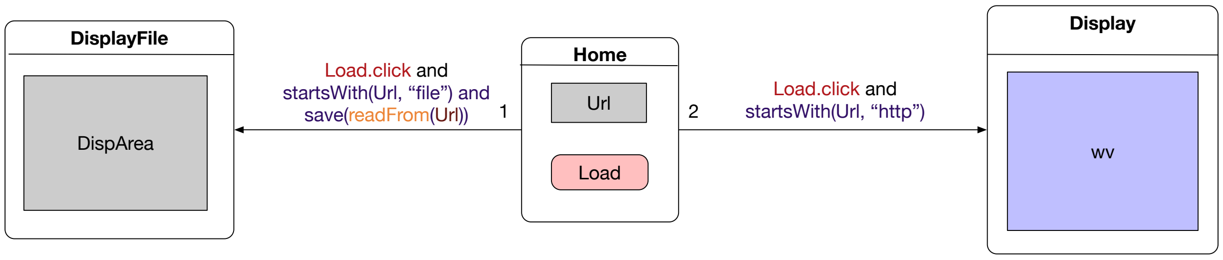


Figure 5.15: *Data Leak Example: Adding boolean constraints*

`save` operation uses the `write` capability provided by the pre-defined resource `EXT_STORE` to write data to a file in the device's external storage.

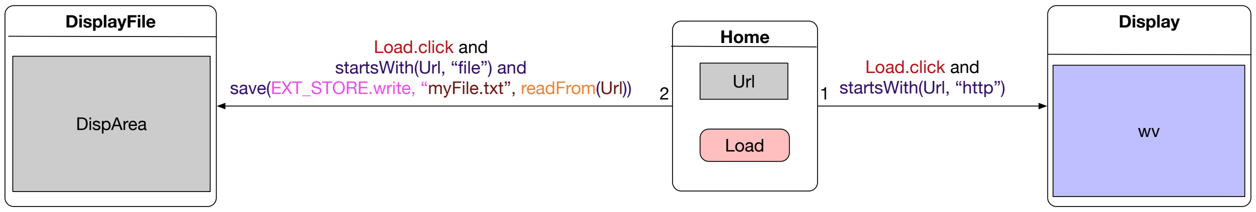


Figure 5.16: *Data Leak Example: Adding boolean constraints*

Connect screen input parameters to data sources: As illustrated in the previous example, a developer provides arguments as data sources to a screen's input parameters. For example, the `Display` screen has an input parameter `u`. The arguments to `u` are either provided by the widget `Url` when the transition from `Home` to `Display` is taken (in green in Figure 5.17 or by an external app (in purple in Figure 5.17).

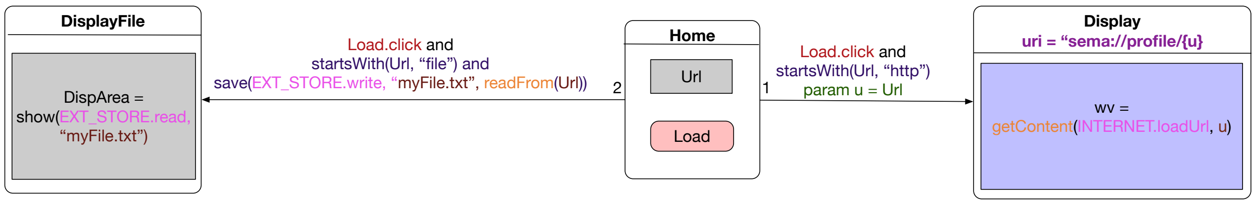


Figure 5.17: *Data Leak Example: Input Parameters, URIs, and Non-boolean operations*

Connect widgets to data sources: A developer provides the data sources that will be used by widgets to display data to users. As shown in Figure 5.18, the non-boolean operations `show` and `getContent` are used as data sources of `Display` and `wv` widgets respectively. The operation `show` reads a file in the device's external storage and the operation `getContent` loads web content from a URL.

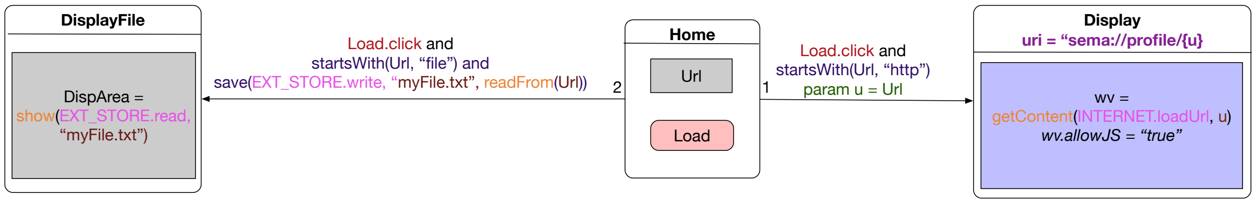
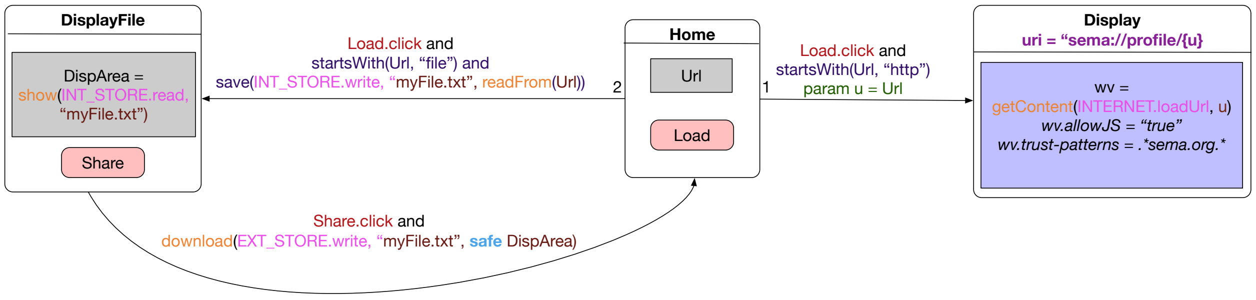

DisplayFile
DispArea = show(EXT_STORE.read, "myFile.txt")
Load.click and startsWith(Url, "file") and save(EXT_STORE.write, "myFile.txt", readFrom(Url))
2
Home
Url
Load
1
Load.click and startsWith(Url, "http") param u = Url
Display
uri = "sema://profile/{u}
wv = getContent(INTERNET.loadUrl, u)
wv.allowJS = "true"

Figure 5.18: *Data Leak Example: Widget Value*

Configure WebView : A developer configures the `Webview` widget to allow JavaScript execution, which is done by setting the `allowJS` flag to "true" as shown in Figure 5.18.

Analyze for security violations: A developer analyzes the specified behavior in the storyboard to check for violations related to confidentiality and integrity. The analysis reveals a violation of confidentiality due to a data leak vulnerability. If the file path provided in `Url` is a file in the app's internal storage, then any file from the app's internal storage can be stored in the device's external storage. Thus, all apps installed in the device can access the app's internal files since files in external storage can be accessed by all apps. This violates the confidentiality of the data read from trusted sources. Information flow analysis will detect and flag this violation by following the chain of flow from sources to untrusted sinks. Further, rule checking will flag a violation of Rule 2 because a whitelist of trusted URLs is not specified for `wv` widget via `trust-patterns` attribute. The absence of whitelist will allow `wv` to execute JavaScript embedded in potentially malicious URLs.

Apply Fix suggested by SeMA: One way to fix this vulnerability is to not save internal files to external storage. Instead, the data in an internal file should be directly displayed in `DisplayFile`. The user should be provided with a button to download the displayed data in external storage. This design is more secure since data in the app's internal files cannot be exposed without the user's explicit approval via a UI action (e.g., button click). The rule violation is fixed by specifying a whitelist of trusted URLs for `wv` (e.g., `wv.trust-patterns=` `{".*sema.org.*"}`). A secure design of the app is shown in Figure 5.19.



DisplayFile
DispArea = show(INT_STORE.read, "myFile.txt")
Share
Load.click and startsWith(Url, "file") and save(INT_STORE.write, "myFile.txt", readFrom(Url))
2
Home
Url
Load
1
Load.click and startsWith(Url, "http") param u = Url
Display
uri = "sema://profile/{u}
wv = getContent(INTERNET.loadUrl, u)
wv.allowJS = "true"
wv.trust-patterns = .*sema.org.*
Share.click and download(EXT_STORE.write, "myFile.txt", safe DispArea)

Figure 5.19: *Data Leak Example: Secure Version*

5.5 Implementation

Android JetPack Navigation (AJN) is a suite of libraries that helps Android developers design their apps' navigation in the form of navigation graphs. A navigation graph is a realization of a traditional storyboard in Android Studio. I have extended navigation graphs with features that enable developers to specify an app's storyboard, as illustrated in Figure 5.4, in Android Studio. The developer can visually represent the screens, widgets, and transitions in the navigation graph. While a developer cannot specify operations and constraints visually, she can specify them in the corresponding XML structure of the navigation graph.

I have extended Android Lint [123], a static analysis tool to analyze files in Android Studio, to implement the analysis and verification of security properties. The analysis is packaged as a Gradle Plugin that can be used from Android Studio.

I have implemented a code generation tool that takes a navigation graph and translates it into Java code for Android. A developer can extend the generated code with business logic in Java or Kotlin. The code generation tool is also packaged as a Gradle Plugin that can be used from Android Studio.

Designing an app in Android Studio with SeMA An app developer uses the existing AJN libraries to build a navigation graph of the app. This graph serves as the initial storyboard (e.g., Figure 5.1). Next, she uses the SeMA provided extensions to the navigation graph to iteratively specify the app's behavior. In each iteration, she uses the extensions to Android Lint tool to analyze the navigation graph for violations of pre-defined security properties. Once the developer has verified the storyboard satisfies the desired properties, she uses the code generation tool to generate an implementation of the app. Finally, she adds the business logic to the generated implementation to complete the implementation of the app.

All thru the development process, the developer operates within the existing development environment while exercising the extensions provided by SeMA.

5.6 Evaluation

I evaluated SeMA in terms of (1) *Effectiveness* focused on its ability to detect and prevent vulnerabilities, and (2) *Usability* focused on its ability to help developers build apps with features that are often found in real-world apps but without a set of known Android app vulnerabilities.

5.6.1 Effectiveness

I used the Ghera benchmark suite, described in Chapter 2, for this evaluation. Ghera has 60 benchmarks. Each benchmark captures a unique vulnerability. I used Ghera because the vulnerabilities in Ghera benchmarks are *valid* (i.e., they have been previously reported in the literature or documented in Android documentation). These vulnerabilities are *general and exploitable* as they can be verified by executing the corresponding benchmarks on vanilla Android devices and emulators. Further, each vulnerability is *current* as they are based on Android API levels 22 thru 27, which enable more than 90% of Android devices in the world and are targeted by both existing and new apps. Finally, the benchmarks are *representative* of real-world apps in terms of the APIs they use, as established in Chapter 3. Hence, the benchmarks in Ghera are well-suited for this evaluation.

For each Ghera benchmark, I used SeMA to create a storyboard of the *Benign* app. If SeMA detected the vulnerability in *Benign's* storyboard, then I modified the storyboard till SeMA found no vulnerabilities. I generated the code from this storyboard and verified the absence of the vulnerability captured in *Benign* by executing it with the *Malicious* app. If the *Malicious* app was unable to exploit the generated app, then I deemed that SeMA prevented the vulnerability.

SeMA prevented 49 of the 60 vulnerabilities captured in Ghera benchmarks. Of the 49 prevented vulnerabilities, 30 were prevented by information flow analysis and rule checking of the storyboard. The remaining 19 were prevented by code generation. Table 5.12 provides the breakdown of the vulnerabilities in Ghera prevented by SeMA.

Of the 30 vulnerabilities detected by the analysis on the storyboard, 15 were prevented by information flow, nine were prevented due to rule analysis, and six were prevented by a combination of information flow and rule analysis. Two of these six vulnerabilities were prevented by rule analysis and could have been prevented by code generation. These two vulnerabilities relate to connecting to HTTP remote servers and connecting to HTTPS remote servers without certificate pinning. While *such vulnerabilities can be prevented by code generation, I chose rule checking in the current realization of SeMA to offer flexibility in using HTTP vs. HTTPS and certificate pinning in storyboards when connecting to remote servers.*

The current realization of SeMA was not applicable to 11 benchmarks in Ghera. While the capabilities in SeMA are equipped to handle eight of the 11 benchmarks, the current realization of SeMA does not yet have support for developing apps with features that cause vulnerabilities in the eight benchmarks (e.g., Content Providers).

The remaining three benchmarks capture vulnerabilities that cannot be prevented by the methodology since they occur due to defects in implementation and are not introduced while specifying a storyboard (e.g., using a library containing vulnerabilities).

Comparison with existing efforts in vulnerability detection. Of the 49 vulnerabilities prevented by the methodology, 28 can be detected curatively by source code analysis after implementing the apps. *Detecting the remaining 21 vulnerabilities by source code analysis is harder* due to combinations of factors such as the semantics of general-purpose programming languages (e.g., Java), security-related specifications provided by the developer (e.g., source/sink APIs), and the behavior of the underlying system (e.g., Android libraries and runtime). This observation is supported by the results in Chapter 4 that show that existing source code analysis tools are not effective in detecting vulnerabilities in an earlier version of Ghera, which included 15 of the 21 vulnerabilities. Further, Pauck et al. [52] evaluated six prominent static taint analysis tools aimed to detect data leak vulnerabilities in Android apps and discovered that most tools detect approximately 60% of the vulnerabilities captured in the DroidBench 3.0 benchmark suite. Finally, Luo et al. [124] qualitatively analyzed Android app static taint analysis tools and observed that these tools need to be carefully configured (e.g., relevant source/sink APIs) and should consider application context to detect vulnerabilities in Android apps accurately.

In Chapter 4, 14 security analysis tools in isolation could detect at most 15 vulnerabilities and the full set of tools collectively detected 30 vulnerabilities. This result suggests combining different analysis will likely be more effective in detecting vulnerabilities. My experience with SeMA suggests *the same is likely true in the context of preventing vulnerabilities: a combination of information flow analysis (deep), rule checking (shallow), and code generation (shallow) helped detect and prevent 49 vulnerabilities.*

Gadient et al. [125] found that real-world apps often expose credentials (e.g., crypto keys) in the source code, use insecure communication channels (e.g., HTTP), and use malicious input to load URLs in a WebView. These vulnerabilities are also captured in Ghera and were prevented by SeMA in this evaluation. This finding suggests *there are non-trivial opportunities for techniques like SeMA to help improve security of real-world apps.*

Table 5.12: Results showing how a vulnerability in a benchmark was detected/prevented. CG, IF, and RC refer to Code Generation, Information Flow Analysis, and Rule-based Analysis, respectively.

ID	Benchmark	Method
C1	BlockCipher-ECB-InfoExposure (A.1.1)	CG
C2	BlockCipher-NonRandomIV-InfoExposure (A.1.2)	CG
C3	ConstantKey-Forgery (A.1.3)	RC
C4	ExposedCredentials-InfoExposure (A.1.4)	CG
C5	PBE-ConstantSalt-InfoExposure (A.1.5)	CG
I1	DynamicBroadcast-UnrestrictedAccess (A.2.1)	RC
I2	EmptyPendingIntent-PrivEscalation (A.2.2)	IF
I3	FragmentInjection-PrivEscalation (A.2.3)	IF
I4	HighPriority-ActivityHijack (A.2.4)	IF
I5	ImplicitPendingIntent-PrivEscalation (A.2.5)	IF
I6	IncorrectHandlingImplicitIntent-UnauthorizedAccess (A.2.7)	IF
I7	NoValidityCheckOnBroadcast-UnintendedInvocation (A.2.8)	RC
I8	OrderedBroadcast-DataInjection (A.2.9)	IF
I9	UnprotectedBroadcastRecv-PrivEscalation (A.2.16)	RC
I10	TaskAffinity-ActivityHijack (A.2.12)	CG
I11	TaskAffinity-LauncherActivity-PhishingAttack (A.2.13)	CG
I12	TaskAffinity-PhishingAttack (A.2.11)	CG
I13	TaskAffinityAndReparenting-PhishingAndDoSAttack (A.2.14)	CG
N1	CheckValidity-InfoExposure (A.3.1)	CG
N2	IncorrectHostNameVerification-MITM (A.3.2)	CG

Continued on next page

Table 5.12 – *Continued from previous page*

ID	Benchmark	Method
N3	InsecureSSLSocket-MITM (A.3.3)	CG
N4	InsecureSSLSocketFactory-MITM (A.3.4)	CG
N5	InvalidCertificateAuthority-MITM (A.3.5)	CG
N6	OpenSocket-InfoLeak (A.3.6)	IF
N7	UnEncryptedSocketComm-DataInjection (A.3.7)	IF
N8	UnPinnedCertificate-MITM (A.3.8)	RC
P1	UnnecesaryPerms-PrivEscalation (A.5.1)	CG
P2	WeakPermission-UnauthorizedAccess (A.5.2)	RC
S1	ExternalStorage-DataInjection (A.6.1)	IF
S2	ExternalStorage-InformationLeak (A.6.2)	IF
S3	InternalStorage-DirectoryTraversal (A.6.3)	IF
S4	InternalToExternalStorage-InformationLeak (A.6.4)	IF
Y1	CheckCallingOrSelfPermission-PrivilegeEscalation (A.7.1)	CG
Y2	CheckPermission-PrivilegeEscalation (A.7.2)	CG
Y3	EnforceCallingOrSelfPermission-PrivilegeEscalation (A.7.5)	CG
Y4	EnforcePermission-PrivilegeEscalation (A.7.6)	CG
Y5	ClipboardUse-InformationExposure (A.7.3)	IF
Y6	DynamicCodeLoading-CodeInjection (A.7.4)	IF
Y7	UniqueIDs-IdentityLeak (A.7.7)	IF
W1	WebView-CookieOverwrite (A.8.3)	IF & RC
W2	HttpConnection-MITM (A.8.1)	RC
W3	JavaScriptExecution-CodeInjection (A.8.2)	IF & RC
W4	UnsafeIntentURLImpl-InformationExposure (A.8.4)	IF & RC

Continued on next page

Table 5.12 – *Continued from previous page*

ID	Benchmark	Method
W5	WebViewAllowContentAccess-UnauthorizedFileAccess (A.8.6)	IF & RC
W6	WebViewAllowFileAccess-UnauthorizedFileAccess-Lean (A.8.7)	IF & RC
W7	WebViewIgnoreSSLWarning-MITM (A.8.8)	CG
W8	WebViewInterceptRequest-MITM (A.8.9)	RC
W9	WebViewLoadDataWithBaseUrl-UnauthorizedFileAccess (A.8.10)	IF & RC
W10	WebViewOverrideUrl-MITM (A.8.11)	RC

5.6.2 Usability

While the evaluation with the Ghera benchmarks shows that SeMA can be used to prevent known vulnerabilities in small apps, it does not tell us if SeMA can be used by developers to uncover vulnerabilities in apps with real-world capabilities and features. Further, SeMA extends storyboards, an existing design artifact, to enable formal reasoning of security properties at design time. While formal reasoning approaches have been proven to be effective in terms of uncovering defects in software (e.g., bugs and vulnerabilities), they can be a burden on developers in terms of time to learn and use [126]. Consequently, the adoption of such approaches in domains like mobile app development may be limited.

To address the concerns identified above, I conducted a usability study of SeMA with ten developers and 13 real-world apps. In this study, I will answer the following research questions:

- **RQ1**: Does SeMA affect app development time?
- **RQ2**: Does software development experience affect development time while using SeMA?

- **RQ3**: Does security-related feature development experience affect development time while using SeMA?
- **RQ4**: Do features used in an app affect development time while using SeMA?
- **RQ5**: Does SeMA detect specific (expected) vulnerabilities introduced in an app's storyboard?
- **RQ6**: Does the use of SeMA (instead of the usual Android app development process) increase the likelihood of introducing expected vulnerabilities?
- **RQ7**: Does software development experience affect the vulnerabilities introduced while using SeMA?
- **RQ8**: Does security-related feature development experience affect the vulnerabilities introduced while using SeMA?

In the following sections, I will describe the different aspects of the experiment, explain the various design decisions, and present the results of the analysis on the observed data.

Study Design

I designed an experiment to gain insights into the effect of introducing SeMA to the Android app development process. Specifically, I wanted to determine if SeMA helps a developer prevent vulnerabilities when making an app. Additionally, I wanted to find out the cost of using SeMA in terms of time taken to build an app with SeMA. Hence, I carried out the following tasks to accomplish the study:

1. Identify real-world Android apps with expected vulnerabilities.
2. Hire developers to participate in the study.
3. Conduct interventions for the developers participating in the study.
4. Create development exercises for the developers participating in the study.

5. Assign exercises to developers.

6. Collect observational data during exercises.

7. Design and administer surveys to collect information about developer experience and developer opinion on SeMA.

8. Analyze collected data.

In the following paragraphs, I will explain the details and the rationale for performing the above tasks.

Identifying Real-world Apps: For this experiment, I selected 30 Ghera benchmarks. The selection was guided by the possibility of detecting the captured vulnerabilities via information flow analysis or rule checking. These vulnerabilities are representative of vulnerabilities found in real-world apps based on API usage information, as shown in Chapter 3. Hence, they are appropriate for measuring the effectiveness of the SeMA methodology.

Next, I randomly collected 50 apps from Google Play, Android's official app store. I manually analyzed the source code of the apps to look for features used in the 30 Ghera benchmarks. The first 13 apps I analyzed accounted for the features used by 26 of the 30 selected Ghera benchmarks. None of the remaining 27 apps used features used by the remaining 4 of the selected Ghera benchmarks. Of the 13 apps, seven apps had been publicly reported to have at least one of the 30 Ghera vulnerabilities [57]. Based on the used features, each of the 13 apps were associated with at least two, on average six, and at most 12 vulnerability benchmarks. The associated number of benchmarks hints at the number of vulnerabilities to expect if these apps were recreated. I refer to these vulnerabilities as expected vulnerabilities from hereon. Table 5.13 provides a brief description of each app and lists the expected vulnerabilities in each app.

Hiring Developers: Since the purpose of this evaluation is to measure the effect of SeMA on Android app development, the obvious candidates for this study are Android app developers. Due to limited local population of developers familiar with Android app development,

ID	App Name	App Description	Expected Ghera Vuln.	Total
1	IRS2Go	Allows registered users to check tax refund status and make tax-related payments	I4,N8,S1,S2,W1	5
2	Geico	Allows registered users to view and download their auto insurance policies	N8,S1,S2,Y7,W1	5
3	Slack	Allows registered users to chat and call other registered users	I3,I4,I6,N6,N7,N8, W2	7
4	Ancestry	Allows registered users to track and save their family history details	N8,S2,W2	3
5	MyBlock H&R	Allows registered users to upload tax-related documents and estimate their annual tax	I3,I4,I6,N8,S1,S3, W2	7
6	AESCrypto	Allows users to encrypt and decrypt messages with a password	C3,S1	2
7	Grab	Allows users to book transportation and accommodation at a location	W1,W3,W4,W6, W8,W9,W10	7
8	Zomato	Allows users to order food from restaurants near their location	I6,W1,W3,W4,W6, W8,W9,W10	8
9	Clipboard Manager	Allows users to take notes and share them via the clipboard or save to a remote server	N8,Y5,W2	3
10	IRCCloud	Allows users to upload files in their device to the app	I6,S1,S2,S3,S4	5
11	Harvest	Allows users to uploads bills and receipts of their monthly expenditure	I3,I4,S1,S2,S3,S4	6
12	Firefox	Allows users to view web or file content in a custom browser	S2,S4,W1,W3,W4, W6,W8,W9,W10	9
13	Yandex	Allows users to view web content in a custom browser and report a browser crash	I1,I7,I8,I9, P2,W1,W3,W4, W6,W8,W9,W10	12

Table 5.13: *A description of the 13 apps along with the expected Ghera vulnerabilities in each app.*

I reached out to local developers with some professional software development experience. I curated a list of 30 professional developers sourced from personal contacts and sent out personalized emails to each one of them, asking them about their interest in participating in the study. In these emails, I promised to provide a small financial incentive (e.g., gift card) if they participated in the study. Of the 30 developers, 15 agreed to participate in the study. While I started the study with 15 participants, five participants withdrew from the study in the middle. So, I completed the study with ten participants.

The participants have an average of 20 months of software development experience with a minimum of three months and a maximum of 53 months as shown in Table 5.14. Seven of them have experience in developing web applications professionally (i.e., outside of learning environments such as classrooms or boot camps), and four of the seven participants experienced in web development have 12 months of experience or more. Further, four of ten participants have experience in developing security features (e.g., authentication) for applications. Finally, in terms of programming languages, eight of ten participants use Java frequently, followed by JavaScript and Python, both of which are used by four participants. However, only 3 participants have experience developing mobile apps professionally, and none of them were in Android. Hence, participants selected in this study are not expert Android app developers.

The rationale behind selecting professional developers with little to no exposure to Android app development was that professional developers are likely to have real-world software development experience. Further, in general, professional developers are likely to learn quickly since many of them have to learn new concepts, tools, and languages on the job. Finally, since commercial software often has to meet security standards, such developers will have a better understanding of security-related features than beginners. Hence, *considering the experience of the participants, their exposure to security-related feature development, and the popularity of the languages also used for Android app development the selected group of participants are reasonable proxies for a developer involved in developing Android apps with security-related features.*. Finally, considering this set of developers will be more representative of Android app developers than considering students, which is the norm in usability

Participant	Software	Web	Mobile	Android	Security
404	53	17	0	0	17
704	3	3	0	0	0
1004	10	0	8	0	0
1503	14	14	0	0	14
1803	30	18	6	0	12
1904	6	6	0	0	0
2103	12	0	0	0	0
2403	3	0	2	0	0
2703	48	12	0	0	10
3103	6	3	0	0	0

Table 5.14: *Experience in months for each participant across different platforms and security features.*

studies in software engineering [127–129].

Conducting Interventions: Since the participants in the study did not know how to make an Android app, I had to train them in Android app development. Hence, I invited all participants to a single 8-hour session. In this session, I introduced them to the fundamental aspects of an Android app (e.g., activity, intents). This introduction included a presentation of the necessary concepts, secure coding guidelines, and a live demonstration of how to develop an Android app using Android Studio. After this session, the participants were given Android-related resources such as the Android documentation and free video tutorials (e.g., Marakana Android tutorials [130]) to learn more about the APIs needed to build Android apps. They were given five days to examine the additional materials independently. After this brief study period, I conducted a group Q&A session with all the participants. This session comprised two parts. The first part of the session was aimed at helping the participants address any confusion they had about developing an app in Android. The second part of the session was meant to provide them with hands-on experience of Android Studio and Android app development in general.

I designed a similar schedule to train the participants in developing Android apps with SeMA. All the participants were invited to a single 8-hour group session. In this session, I introduced the features in SeMA and demonstrated how to build an app with those features.

I provided the participants with the documentation to SeMA and five days to peruse the materials. Finally, I conducted a group Q&A session with all participants to help them learn about developing an app with SeMA. In addition to the Q&A, in this session, they made an app with SeMA with our assistance. The hands-on experience with SeMA helped them get acclimatized to Android app development with SeMA.

This entire process of teaching ten participants Android and SeMA took approximately two weeks to complete.

In any usability study that requires interventions, the application of the interventions to the users should be designed carefully. In this study, since the training received by a participant could affect the results, *it was necessary to ensure that they received homogeneous instruction*. Hence, I conducted all the training and Q&A sessions for the participants together instead of individually. However, since each participant was allowed to independently explore Android and SeMA, the effect of individual learning will affect the results of this experiment.

Creating Development Exercises: I analyzed the selected 13 real-world apps in two ways to design the development exercises for the participants. First, I decompiled the app and manually analyzed its metadata and source code. This analysis revealed the screens in the app, any external apps (e.g., remote server) the app communicates with, the data flow of the app, and the APIs used in the apps. Second, I installed the app on an Android device and interacted with it to understand the navigation of the app. Based on the analysis of an app's innards and navigation, I constructed a specification of the app. These specifications were used as exercises for the participants in the study.

The motivation behind using these specifications as exercises for participants was that I wanted to mimic a real-world scenario where a developer would have to start from a description of an app's expected behavior, understand it, and use the relevant features in Android to realize the app. Further, since one of our goals was to measure the effectiveness of SeMA in helping developers uncover vulnerabilities, the specifications provided in the exercises were constructed to force participants to make decisions that would affect the security of the app.

As an example, consider the snippet from an exercise assigned to a participant in this experiment – *"When the app is started, the user is shown a FileUploader screen. The FileUploader screen has a button to allow users to upload a file. Upon clicking this button, the user is shown a list of files. Upon selecting a file, the selected file is saved and the user is shown a message to indicate if the file was saved successfully."*. This specification asks the participant to design an app that allows users to upload a file, a feature that requires the participant to consider security implications. For example, if the participant decided to use an external client to help the user select a file, then she would register the app to receive the chosen file path from the external client via a message. In doing so, an app might open itself to accepting malicious messages. If the app used the file path embedded in a malicious message to create a destination where the selected file will be uploaded, then this decision would enable data injection or directory traversal attack. Hence, the approach used by the participant to design the file upload feature will have an impact on the security of the app. These decisions and SeMA's influence on them was the focus of this study.

Assigning Exercises to Participants: I assigned three distinct apps to every participant. Of the three apps assigned to each participant, one baseline app was common to all participants (App ID 1 in Table 5.13). All participants developed the baseline app first with SeMA and then without SeMA (i.e., using the existing Android development process). I ensured that the baseline app with SeMA and without SeMA was developed on different days to reduce the effect of memorization. The rationale for developing the baseline app with SeMA before developing the same app without SeMA was to avoid the inflation of SeMA's effect due to a participant's prior knowledge about the app/exercise.

In addition to the baseline app, each participant built two apps using SeMA. This assignment strategy resulted in 40 sample implementations of 13 apps. Of the 40 samples, 20 were realizations of the baseline app – 10 using SeMA and 10 without SeMA. Keeping the baseline app constant across all participants was necessary to measure the effect of SeMA on one app for all participants. The remaining 20 samples were uniformly distributed across the remaining 12 apps (see Table 5.15). This distribution ensured all expected vulnerabilities could be introduced by a participant.

Observing Development and Collecting Data: Each participant worked on the assigned development exercise via remote session. Participants decided on the length of each session. There was no limit on the number of sessions to work on a development exercise.

In every session, participants shared their screen with us. Via this screen sharing, I observed the development of apps and recorded when participants introduced vulnerabilities that were expected in the developed apps. Also, I recorded the error messages and causes of error reported by SeMA.

Designing and Administering Development Experience Surveys: I administered two surveys after the participants finished their exercises. The purpose of the first survey was to build a participant profile. It asked about the time spent developing software outside formal learning environments (e.g., classrooms), experience developing web and mobile apps, and familiarity with different programming languages and technologies. The purpose of the second survey was to assess their experience with SeMA. It asked about various features of SeMA and if they aided/impeded the development workflow.

Analyzing Observational Data: To answer RQ1, I analyzed the development time of developing the baseline app with and without SeMA. The development of baseline apps without SeMA served as the control group, while the development of baseline apps with SeMA served as the treatment group. I computed the average difference in app development time between the control and treatment groups and used a paired two-tailed t-test to measure the effect of SeMA on app development time.

To answer RQ2, I partitioned the participants into two groups. The +2DX group comprised participants with two or more years of software development experience and the -2DX group comprised participants with less than two years of development experience. I used a two sample two-tailed t-test to compare the average development time using SeMA between

	AppID											
Participant	2	3	4	5	6	7	8	9	10	11	12	13
404			102(3)			85(2)						
704								105(2)		135(3)		
1004											55(2)	
1503	150(2)				180(3)							
1803				240(3)				90(2)				
1904		150(2)										110(3)
2103					105(3)				150(2)			
2403		120(2)									60(3)	
2703							105(2)			132(3)		
3103									100(3)			80(2)

Table 5.15: *Time taken (in minutes) by a participant to make non-baseline apps. The number in brackets indicates if an app was a participant's 2nd or 3rd app.*

these groups to quantify the effect of software development experience on developing software using SeMA.

To answer RQ3, similar to RQ2, I partitioned the participants into two groups. +1SFDX group comprised of participants with one or more years of security feature development experience and -1SFDX group comprised of participants with less than one year of security feature development experience.

To answer RQ4, I compared the average time taken by participants using SeMA to make apps with expected vulnerabilities stemming from the features of a category and apps without the same features. I used a two sample two-tailed t-test for this comparison.

To answer RQ5, I measured the proportion of total number of expected vulnerabilities introduced by all participants while using SeMA that were then successfully detected by SeMA. I also considered the proportion of expected vulnerabilities that were prevented by SeMA (due to good defaults).

While SeMA is effective in detecting expected vulnerabilities, its use could affect the introduction of expected vulnerabilities. I tackle this concern in RQ6 — *does the use of SeMA (instead of the usual Android app development process) increase the likelihood of introducing expected vulnerabilities?*

To answer RQ6, I measured the average proportion of expected vulnerabilities introduced by participants in the baseline app with and without SeMA. I compared the average proportions with a paired two-tailed t-test to determine if there was any significant difference in the average proportion of vulnerabilities introduced in the baseline app with and without SeMA.

To answer RQ7, I computed the proportion of expected vulnerabilities introduced while using SeMA for both +2DX and -2DX groups. I compared these two proportions using a two sample two-tailed z-test to measure the effect of the participants' software development experience on introducing vulnerabilities with SeMA.

To answer RQ8, I compared the proportion of expected vulnerabilities introduced while using SeMA by participants in +1SFDX and -1SFDX group. I used a two sample two-tailed z-test for the comparison.

Results from Observational Data

RQ1: Does SeMA affect app development time? Every participant took less time to make the baseline app with SeMA compared to without SeMA, as shown in columns 2 and 3 in Table 5.16. The median development time of an app with SeMA is 188 minutes and without SeMA is 283 minutes, which shows that, for the participants considered in this study, *developing the baseline app with SeMA took 40% less than developing the same app without SeMA*.

Based on paired two-tailed t-test, SeMA has a significant effect (p-value = 0.0002 with 95% confidence interval [79 mins., 177 mins.]]) on the mean development time in developing the baseline app. The 95% confidence interval of SeMA's effect on decreasing the development time suggests that SeMA has the potential to reduce app development time.

RQ2: Does software development experience affect development time while using SeMA? The three participants in the +2DX group took 142 minutes on average, to develop an app with SeMA. This time was 150 minutes for the seven participants in the -2DX group. Based on two sample two-tailed t-test comparing the means of these two groups, there is no significant difference in the average time taken by participants from these two groups while using SeMA (p-value = 0.75 with 95% confidence interval [-55 mins., 39 mins.]). Hence, *software development experience (not related to Android app development) does not affect the time taken by a developer to make an app with SeMA*.

RQ3: Does security-related feature development experience affect development time while using SeMA? The four participants in the +1SFDX group took 160 minutes on average, to make an app with SeMA. This time was 139 minutes for the six participants in the -1SFDX group. While participants in the +1SFDX group took more time than participants in the -1SFDX group in this sample, the difference is not significant based on two sample two-tailed t-test (p-value = 0.37 with 95% confidence interval [-28 mins., 70 mins.]). Hence, *security-related feature development experience does not affect the time taken by a developer to make an app with SeMA*.

Participant	Time to Baseline App no SeMA	Time to Baseline App SeMA	Time to 2nd App SeMA	Time to 3rd App SeMA	Expected Vulns.	Added Vulns.
404	276	165	85	102	15	10
704	440	255	105	135	14	4
1004	180	140	55	135	22	10
1503	588	315	180	150	12	5
1803	360	210	90	240	16	7
1904	290	210	150	110	24	16
2103	390	210	150	105	12	5
2403	210	138	120	60	21	12
2703	240	150	105	132	19	11
3103	240	135	80	100	22	11
Average	321	193	112	138		
95% C.I		[124,170]				

Table 5.16: *Observational Data of each participant. The unit of time is minute. Only the baseline app was made with and without SeMA. The non-baseline apps were made only with SeMA. Expected Vulns. indicates no. of vulnerabilities that could have been introduced by a participant while developing the three apps assigned to her with SeMA. Added Vulns. indicates the no. of vulnerabilities introduced by a participant while developing the three apps assigned to her with SeMA.*

RQ4: Do the features used in an app affect development time while using SeMA? With SeMA, for the seven apps with ICC features based expected vulnerabilities (ICC apps), the average development time was 58 minutes more than the average development time of the six apps without ICC-based expected vulnerabilities (non-ICC apps). Based on two sample two-tailed t-test, the 95% confidence interval of the difference between the average development time to make ICC apps and non-ICC apps ranges from 21 to 95 minutes. A likely reason for longer development time for ICC-apps is that ICC-apps tend to have more screens and navigation between the screens. Considering the significant difference in development time for at least one set of features (e.g., ICC), *the features used in an app are likely to affect the time taken to make the app with SeMA*.

Based on the average development time using SeMA, the development of baseline apps took more time than the development of non-baseline apps (see the *average* row in Table 5.16). The longer development time for the baseline app is likely due to the methodology's novelty, considering the baseline app was the first app participants developed using SeMA. This observation suggests that development time reduced as participants became familiar with SeMA. However, development time did not necessarily decrease consistently across all apps made with SeMA. For example, participant 1803 took more time to develop the third app than the second app. This increase was likely because the third app had more features than the second app, making the third app more complicated. Specifically, the third app was more complicated since it had features related to user registration and authentication, which were absent in the second app. Hence, *the reduction in development time with SeMA seems to be dependent on the familiarity with SeMA and the features used in the app.*

RQ5: Does SeMA detect specific (expected) vulnerabilities introduced in an app's storyboard? The participants in this study introduced 91 of 177 instances of expected vulnerabilities in 30 app storyboards while using SeMA (see *Added Vulns.* column in Table 5.16). For example, participants introduced all instances of expected vulnerabilities in the Web category (see Table 5.18). SeMA detected and reported a violation for every one of the 91 instances. Hence, *SeMA is highly likely to detect expected vulnerabilities introduced*

Participant	N	S	T	W	K	D
404	3	2	5	0.6	0.4	0.2
704	2	0	5	0.4	0	0.4
1004	2	2	5	0.4	0.4	0
1503	2	3	5	0.4	0.6	-0.2
1803	2	2	5	0.4	0.4	0
1904	2	3	5	0.4	0.4	0
2103	1	1	5	0.2	0.2	0
2403	2	0	5	0.4	0	0.4
2703	2	1	5	0.4	0.2	0.2
3103	2	1	5	0.4	0.2	0.2
Average				0.4	0.3	0.1

Table 5.17: *Proportion of expected vulnerabilities introduced in the baseline app by a participant with and without SeMA. N indicates the no. of expected vulnerabilities introduced by a participant in the baseline app without using SeMA. S indicates the no. of expected vulnerabilities introduced by a participant in the baseline app using SeMA. T indicates the no. of expected vulnerabilities in the baseline app. W indicates the proportion of expected vulnerabilities introduced by a participant in the baseline app without SeMA. K indicates the proportion of expected vulnerabilities introduced by a participant in the baseline app with SeMA. D is the difference between W and K.*

in a storyboard.

I anticipated SeMA would prevent 38 instances of expected vulnerabilities – two instances of I1, 16 instances of I4, two instances of I9, and 18 instances of N8 – due to good defaults provided by SeMA, e.g., certificate pinning is enabled by default. Since participants overrode the defaults in three instances – two instances of I4 and one instance of I9, 35 of the 38 instances of expected vulnerabilities were prevented by the good defaults in SeMA. Hence, *the defaults built into SeMA are likely to prevent expected vulnerabilities.*

Overall, since SeMA detected and prevented 126 of 177 (71%) instances of expected vulnerabilities in the considered sample, we can conclude *SeMA is likely to detect and prevent expected vulnerabilities in an app's storyboard.*

Participants did not introduce 51 instances of expected vulnerabilities – one in Crypto, five in ICC, 17 in Networking, 27 in Storage, and one in System – while using SeMA. For example, in the Networking category, 15 of 18 instances of W2 were not introduced since participants decided to use HTTPS over HTTP when communicating with a remote server. In Storage category, when participants introduced S2 (writing to external storage), SeMA flagged the vulnerability. Since this informed the participants about the pitfalls of using external storage, they were more cautious with the further use of external storage. Consequently, only 5 of the 19 expected instances of S1 (reading from external storage) vulnerability were introduced by the participants. Hence, *prior knowledge of vulnerabilities is likely to affect the introduction of vulnerabilities and, consequently, the lower the observed effectiveness of SeMA.* On the positive side, unlike passive interventions like lists of vulnerabilities, *SeMA is likely to serve as an active intervention and improve developer awareness about vulnerabilities.*

RQ6: Does the use of SeMA (instead of the usual Android app development process) increase the likelihood of introducing expected vulnerabilities? The average percentage of introducing an expected vulnerability in the baseline app without SeMA was 40% across all participants (column W in Table 5.17). With SeMA, the average percentage of introducing an expected vulnerability in the baseline app was 30% (column K in Table 5.17). The median and mean difference between the percentage of expected vulnerabilities introduced with and without SeMA in the baseline app for each participant is 0% and 10%, respectively (see column D of Table 5.17). Further, based on paired two-tailed t-test, there is no significant difference between the mean percentage of introducing an expected vulnerability in the baseline app with and without SeMA (p-value = 0.17 with 95% confidence interval [-6%, 26%]). Consequently, *SeMA is not likely to introduce any more or less vulnerabilities compared to the prevalent app development process.*

RQ7: Does software development experience affect the vulnerabilities introduced while using SeMA? If we consider the participants' software development experience, then the three participants in the +2DX group introduced 53% of 50 expected vulnerabilities while using SeMA. The seven participants in the -2DX group introduced 50% of 127 expected vulnerabilities while using SeMA. The difference in the percentage of introducing an expected vulnerability between the two groups of participants is not statistically significant (p-value

Category	Benchmark ID	# Apps (A)	# Expected (E)	# Introduced (I)
Crypto	C3	1	2	1
ICC	I1	1	2	0
	I3	2	4	3
	I4	4	16	2
	I6	5	9	9
	I7	1	2	0
	I8	1	2	0
	I9	1	2	1
ICC Ave.		2.1	5.3	2.1
Networking	N6	1	2	1
	N7	1	2	1
	N8	6	18	0
	W2	7	18	3
Networking Ave.		3.8	10	1.25
P2	Permission	1	2	2
Storage	S1	6	19	5
	S2	7	20	10
	S3	3	6	5
	S4	3	6	4
Storage Ave.		4.8	12.8	6
System	Y5	1	2	2
	Y7	1	1	0
System Ave.		1	1.5	1
Web	W1	4	6	6
	W3	4	6	6
	W4	4	6	6
	W6	4	6	6
	W8	4	6	6
	W9	4	6	6
	W10	4	6	6
Web Ave.		4	6	6
Total Ave.		3.1	6.8	3.5

Table 5.18: *Frequency distribution of each Ghera vulnerability. A indicates no. of apps with an expected vulnerability. E indicates no. of times a vulnerability was expected to be introduced. I indicates no. of times an expected vulnerability was introduced and prevented.*

= 0.44 with 95% confidence interval [-10%, 22%]). Hence, *software development experience does not affect the proportion of expected vulnerabilities introduced by a developer in an app while using SeMA.*

RQ8: Does security-related feature development experience affect the vulnerabilities introduced while using SeMA? The four participants with one year or more experience in security-related feature development introduced 54% of 62 expected vulnerabilities while using SeMA. The six participants with less than a year's experience introduced 49% of 115 expected vulnerabilities while using SeMA. Further, there was no significant difference between the proportion of expected vulnerabilities introduced by the two groups of participants (p-value = 0.72 with 95% confidence interval [-13%, 17%]). A possible reason for the absence of a statistically significant difference is, while the participants in the +1SFDX group have prior knowledge of security features, they do not have experience in security feature development in the context of Android apps. Hence, *security-related feature development experience not specific to Android does not affect the proportion of expected vulnerabilities introduced by a developer in an app while using SeMA.*

Observations from Survey Data

As mentioned previously, I conducted a survey to collect responses from participants about their experience using SeMA. In the following paragraphs, I present our observations based on a qualitative analysis of the participants' responses. The participant responses are summarized in Table 5.19

Did the SeMA extension to Android Studio help/impede in Android app development? All participants found the extensions to Android Studio helpful. These extensions included the extensions added to a storyboard (described in Section 5.3.1) and capabilities related to generating code from a storyboard. Seven of the ten participants said that the extensions helped them *a lot*. The remaining three said that the extensions aided their workflow *quite a bit*. On the flip side, all participants, except one, said that the extensions did not impede their development. Even the one exception said that the extensions only impeded *a little*. Hence, *the participants in this study found the SeMA extension to Android Studio to be largely useful.*

Did the property annotation feature of SeMA help/impede in Android app development? Eight of the ten participants found that the property annotations in a storyboard (e.g., constraints on transitions) helped them in Android app development *a lot*. The other two felt that the annotations helped them *quite a bit*. Only one of the ten participants said that the annotations impeded her development *a little*. *The participant responses show that property annotations largely aided in app development and did not impede their workflow.*

Did the property checking feature of SeMA help/impede in Android app development? All the participants, except one, said that the property checking feature in SeMA helped them *a lot*. The one exception said that it helped her *quite a bit*. However, five participants said that it impeded their development *a little* and *quite a bit* while the remaining five said that the property checking feature did not impede at all. Since the property checking feature in SeMA forces developers to fix property violations before moving to the next step, a few participants found it to be impeding their workflow. Hence, *while property checking was helpful for most participants; there is room for improving the process of reporting and fixing violations.*

Did the pre-defined properties provided by SeMA help/impede in Android app development? Six of the ten participants felt that the pre-defined properties (e.g., resources) helped their development *a lot*, and four of them felt that they helped *quite a bit*. In a similar vein, six of them found that the pre-defined properties did not impede their workflow at all, and four of them said that they impeded *a little*. Hence, *participants felt that the pre-defined properties largely aided their workflow.* The four participants who found the pre-defined properties to be *a little* invasive said that it was because they had to keep referring to the documentation for understanding how to use them. This problem can be

Question	None	A little	Quite a bit	A lot
		Scale		
How much did the SeMA extension to Android Studio *help* in Android app development?	0	0	3	7
How much did the SeMA extension to Android Studio *impede* in Android app development?	9	1	0	0
How much did the property annotation feature of SeMA *help* in Android app development?	0	0	2	8
How much did the property annotation feature of SeMA *impede* in Android app development?	9	1	0	0
How much did the pre-defined properties provided by SeMA *help* in Android app development?	0	0	4	6
How much did the pre-defined properties provided by SeMA *impede* in Android app development?	6	4	0	0
How much did the property checking feature of SeMA *help* in Android app development?	0	0	1	9
How much did the property checking feature of SeMA *impede* in Android app development?	5	5	0	0

Table 5.19: *Questions asked to participants in a survey. The number in each cell indicate the number of participants who chose a particular scale for a question.*

addressed by adding more auto-completion support for SeMA.

Feedback: What would you change in SeMA? All the participants agreed that storyboard-driven development helped them in their development. They particularly appreciated the visualization of the storyboard as it helped them conceptualize the app's behavior. Further, most of them said that SeMA helped them uncover vulnerabilities in the app's design that they would not have otherwise discovered. Typical feedback from participants was to improve the code-completion aid for SeMA and provide access to documentation in the IDE.

Threats to Validity

While the results from this study provide useful insights about the usability and effectiveness of the SeMA methodology, the small number of participants and the small number of apps made by each participant used in this study might affect the generalize-ability of the results. This limitation can be addressed by repeating the experiment with a large number of participants or by increasing the number of apps made by a participant.

Despite the participants in this study having varied experience, the selected participants might not be representative of all kinds of real-world developers. This limitation can be addressed by repeating the study with a more varied sample of developers.

While the participants in this study are reasonable proxies for average Android app developers, they did not have Android app development experience outside of learning environments. Hence, the results and observations from this study might change if repeated with participants with experience in Android app development.

The representation of the Ghera vulnerabilities in the sample of real-world apps selected in this study was not uniform (i.e., some vulnerabilities appeared more than others). This lack of uniformity could have affected the results. This concern can be addressed by repeating the experiment with a different sample of real-world apps.

While the exercises assigned to the participants were based on real-world apps, it is possible that they were influenced by our knowledge of vulnerabilities that occur in Android apps. This influence could have introduced bias for certain vulnerabilities in the exercises. This limitation can be addressed by repeating the experiment with a different set of exercises.

While I ensured that each participant received the same intervention regarding Android app development and SeMA, their personal capacities in grokking new material might have affected the way they made an app. This difference could have influenced the final results.

Finally, I silently observed each participant when they were developing an app. This environment might have caused some participants to behave differently, which might have impacted the way they made an app. The influence of such factors can be verified by conducting studies that consider environment-related aspects.

5.7 Open Challenges

Performance The current realization and evaluation of SeMA has focused on ensuring the correctness of generated code. So, while SeMA adheres to performance guidelines outlined in

Android's documentation in the generated code, the generated code may not be performant. This concern can be verified by evaluating the runtime performance of generated code.

Property Preservation One challenge that remains to be addressed in the current realization of SeMA is ensuring the generated implementation satisfies the security properties satisfied by the storyboard. This hinges on ensuring (1) the integrity of generated code and (2) developer-added business logic does not violate the security properties verified in the app storyboard. The current realization of SeMA deters developers from modifying the generated code. The generated code is kept separate from the developer-added code in Android Studio. If a developer modifies generated code, a warning message is shown to the user as a pop-up. If the developer modifies the generated code in spite of the warning, then the code is re-generated when the developer compiles the app. While these methods discourage the developer from modifying the generated code, they do not prevent it. One way to prevent modifications to generated code is to use techniques (e.g., fingerprinting) to check and enforce the integrity of the generated code.

While SeMA deters the modification of generated code, developers can add business logic code in a way that may not guarantee property preservation. This challenge can be tackled by inhibiting the execution of an app upon detecting the violation of security properties (using techniques such as runtime checks and app sandboxing). However, the current realization of SeMA does not address this concern.

5.8 Artifacts

The current realization of SeMA along with the instructions to build and use it is available in the public repository `https://bitbucket.org/secure-it-i/sema/src/master/`.

The raw data collected during the usability study, along with the statistical tests used to analyze the collected data are available in the public repository `https://bitbucket.org/secure-it-i/sema/src/master/usability-test/`.

5.9 Conclusion

In this chapter, we saw an alternative approach to develop secure Android apps. This approach focusses on preventing vulnerabilities as opposed to the traditional approach of detecting vulnerabilities after they have occurred. To this end, SeMA is a design-based methodology based on Model-Driven development and existing design techniques to help build secure Android apps.

SeMA extends storyboards with features that enable developers and designers to collaborate and specify an app's behavior iteratively, at the same time reason about and verify security properties related to confidentiality and integrity in an app's design. Furthermore, SeMA has code generation support that helps to translate annotated/extended storyboards, specified in SeMA, to an implementation. Developers can enrich the generated implementation with business logic code in Java or Kotlin.

A proof-of-concept realization of SeMA is available for Android Studio. An empirical evaluation of SeMA shows that SeMA can prevent 49 of 60 vulnerabilities captured in the Ghera benchmark suite through a combination of information flow analysis, rule checking, and code generation techniques.

A usability study of SeMA with ten professional software developers shows that SeMA is likely to reduce Android app development time and help developers prevent expected vulnerabilities in their apps.

Chapter 6

Summary and Future Directions

We[1] identified that, despite the focus on Android app security in recent systems security research, there is a lack of detailed information about the manifestation of Android app vulnerabilities. Hence, as a solution, we developed Ghera – an open repository of 60 Android app vulnerabilities. While developing Ghera, we discovered characteristics of vulnerability benchmarks, which are applicable in other contexts as well (e.g., performance). Ghera has been used to 1) measure Android app security analysis solutions and 2) to learn about vulnerabilities that occur in Android apps.

The evaluation of Android app security solutions is based on benchmarks, which are either available or prevalent. Hence, we[2] developed the notion of representativeness of a benchmark based on API usage to measure the likelihood of a vulnerability in a benchmark occurring in real-world apps. We used this metric to measure the representativeness of four benchmark suites used to measure the effectiveness of solutions related to Android app security. The observations from this exploration will help tool developers select benchmarks for evaluation appropriately, and benchmark developers identify gaps in their benchmarks.

The recent focus on Android app security has led to the development of numerous tools and techniques to secure Android apps. However, there has been no comprehensive effort

[1]This was a collaborative effort with Venkatesh-Prasad Ranganath, Aditya Narkar, and Nasik-Nafi Muhammad.

[2]This was a collaborative effort with Venkatesh-Prasad Ranganath.

to evaluate the effectiveness of such tools systematically. Hence, we[3] considered 64 Android security analysis tools and frameworks and evaluated the effectiveness of 14 of them in detecting known vulnerabilities captured in the Ghera benchmarks. We discovered that a tool could detect only 15 vulnerabilities in Ghera, and all tools together could detect only 30 of the vulnerabilities.

Since existing tools and techniques are not effective in detecting all known vulnerabilities that occur in Android apps, there is a scope of an alternative solution to secure Android apps. Hence, we[4] developed SeMA, a design methodology to help build secure Android apps. SeMA is based on Model-driven development and storyboarding – an existing design artifact. As opposed to existing techniques, which take a curative approach to secure Android apps by detecting them after they occur, SeMA helps developers prevent vulnerabilities by helping developers reason about security properties such as confidentiality and integrity at design time. SeMA can prevent 49/60 vulnerabilities in Ghera, which is more than what is detected by the 14 vulnerability detection tools. Further, a usability study with ten professional developers, showed developers take between 10% to 70% less time to make an app with SeMA than without SeMA. In terms of effectiveness, SeMA helps developers prevent vulnerabilities in at least 83% of the apps they develop with SeMA, and the vulnerabilities prevented by SeMA is introduced by developers more or less 50% of the time.

6.1 Future Directions

Metrics and Benchmarks There is a need to continuously and rigorously evaluate Android app security solutions. Hence, future work in this area can proceed in the following ways:

- *Extend Ghera with new benchmarks.* There are two ways to add a benchmark to Ghera – (1) create a variant, based on an existing benchmark that captures a manifestation of the vulnerability in that benchmark, and (2) create a benchmark based on a new

[3]This was a collaborative effort with Venkatesh-Prasad Ranganath.
[4]This was a collaborative effort with Venkatesh-Prasad Ranganath, Torben Amtoft, and Michael Higgins.

vulnerability, i.e., a vulnerability not in any Ghera benchmark. Extending Ghera will involve exploring CVE reports, studying Android APIs, and examining real-world apps.

- *Extend Ghera to iOS.* Most research efforts in mobile app security have been focused on Android since Android is open source, and Android apps are packaged as dex code, which is relatively easier to analyze. Although iOS is the other major mobile platform, comparatively fewer efforts have considered iOS app security. Hence, it would be interesting to use Ghera's methodology to catalog iOS app vulnerabilities as benchmarks systematically.

- *Develop richer metrics to measure representativeness of benchmark.* Representativeness of vulnerability benchmarks is crucial to assessing the associated risks as vulnerabilities more likely to be found in real-world apps will be more likely to be exploited. This dissertation has developed API usage as a metric for representativeness. However, richer metrics based on other aspects (e.g., data and control flow) need to be explored to more accurately measure representativeness.

Secure by design apps SeMA proposes an alternative approach to developing secure mobile apps. It advocates making security a first-class citizen of an app's design by enabling collaboration between developers and designers. While the methodology is feasible and usable to some extent, it can be extended in the following ways:

- *Expressibility* This dissertation has explored SeMA in the context of developing Android apps. It will be worthwhile to explore if the concepts captured in SeMA can be used to develop apps in a platform-independent way. This exploration will include mapping concepts used in other app development platforms to the ones in SeMA and developing code generation schemes for each platform.

- *Collaborative-ness* SeMA claims to enable collaboration amongst app developers and app designers. However, there is no evidence to suggest that SeMA indeed enhances collaboration. Hence, usability studies with teams of app developers and designers will need to be conducted to support this claim.

- *Certification* Since SeMA provides a specification of an app's behavior in the form of a storyboard, it can be used to generate a machine-checkable certificate of an app's behavior. App stores can verify such certificates before accepting apps.

Bibliography

[1] Statista. Android vs ios market share, 2019. Available at https://www.statista.com/statistics/272307/market-share-forecast-for-smartphone-operating-systems/.

[2] Statista. No. of android apps in the google play store, 2019. Available at https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.

[3] Open Handset Alliance. Open handset alliance android, 2019. Available at https://www.openhandsetalliance.com/android_overview.html.

[4] Statista. Global developers in leading app stores, 2019. Available at https://www.statista.com/statistics/276437/developers-per-appstore/.

[5] Rita Francese, Carmine Gravino, Michele Risi, Giuseppe Scanniello, and Genoveffa Tortora. Mobile app development and management: Results from a qualitative investigation. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '17, page 133–143. IEEE Press, 2017. doi: 10.1109/MOBILESoft.2017.33.

[6] R. Balebako and L. Cranor. Improving app privacy: Nudging app developers to protect user privacy. *IEEE Security Privacy*, 12(4):55–58, 2014.

[7] Positive Technologies. Vulnerabilities and threats in mobile applications, 2019. Available at https://www.ptsecurity.com/ww-en/analytics/mobile-application-security-threats-and-vulnerabilities-2019/.

[8] Checkmarx. How attackers could hijack your android camera to

spy on you, 2019. Available at https://www.checkmarx.com/blog/how-attackers-could-hijack-your-android-camera.

[9] Yingjie Wang, Xing Liu, Weixuan Mao, and Wei Wang. Dcdroid: Automated detection of ssl/tls certificate verification vulnerabilities in android apps. In *Proceedings of the ACM Turing Celebration Conference - China*, ACM TURC '19. Association for Computing Machinery, 2019. ISBN 9781450371582. doi: 10.1145/3321408.3326665.

[10] Joydeep Mitra Aditya Narkar and Venkatesh-Prasad Ranganath. Cve-2019-9463, 2019. Available at https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9463/.

[11] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 229–240, New York, NY, USA, 2012. Association for Computing Machinery. doi: 10.1145/2382196.2382223.

[12] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, page 239–252. Association for Computing Machinery, 2011. doi: 10.1145/1999995.2000018.

[13] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 280–291, 2015.

[14] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek. Covert: Compositional analysis of android inter-app permission leakage. *IEEE Transactions on Software Engineering*, 41 (9):866–886, 2015.

[15] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, page 627–638. Association for Computing Machinery, 2011. doi: 10.1145/2046707.2046779.

[16] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 217–228. Association for Computing Machinery, 2012. doi: 10.1145/2382196.2382222.

[17] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. Whyper: Towards automating risk assessment of mobile applications. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, page 527–542, USA, 2013. USENIX Association. ISBN 9781931971034.

[18] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88:67 – 95, 2017. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2017.04.001.

[19] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, 2014. doi: 10.1145/2666356.2594299.

[20] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 1329–1341. Association for Computing Machinery, 2014. doi: 10.1145/2660267.2660357.

[21] Sufatrio, Darell J. J. Tan, Tong-Wei Chua, and Vrizlynn L. L. Thing. Securing android: A survey, taxonomy, and challenges. *ACM Comput. Surv.*, pages 58:1–58:45, 2015.

[22] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 393–407. USENIX Association, 2010.

[23] Lok Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 569–584, Bellevue, WA, 2012. USENIX. ISBN 978-931971-95-9.

[24] Vaibhav Rastogi, Yan Chen, and William Enck. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, page 209–220. Association for Computing Machinery, 2013. doi: 10.1145/2435349.2435379.

[25] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, CASCON '10, page 214–224. IBM Corp., 2010. doi: 10.1145/1925805.1925818.

[26] J. C. S. Santos, K. Tarrit, and M. Mirakhorli. A catalog of security architecture weaknesses. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 220–223, 2017.

[27] IEEE Center For Secure Design. Avoiding The Top 10 Software Security Design Flaws. https://ieeecs-media.computer.org/media/technical-activities/CYBSI/docs/Top-10-Flaws.pdf, 2014. Accessed: 01-Jun-2020.

[28] Eugenia Politou, Efthimios Alepis, and Constantinos Patsakis. Forgetting personal data and revoking consent under the GDPR: Challenges and proposed solutions. *Journal of Cybersecurity*, 4, 2018. doi: 10.1093/cybsec/tyy001. URL https://doi.org/10.1093/cybsec/tyy001. tyy001.

[29] Christian Kurtz, Martin Semmann, and Tilo Böhmann. Privacy by design to comply with gdpr: A review on third-party data processors. In *AMCIS*, 2018.

[30] European Union Agency for Cybersecurity. Privacy and data protection in mobile applications. https://op.europa.eu/s/n8kT, 2018. Accessed: 01-Jun-2020.

[31] Joydeep Mitra and Venkatesh-Prasad Ranganath. Cve-2018-9548, 2018. Available at https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-9548/.

[32] Frischknecht P. Gadient P., Ghafari M. and Nierstrasz O. Security code smells in android icc. *Empirical Software Engineering*, 2018. doi: 10.1007/s10664-018-9673-y. URL https://doi.org/10.1007/s10664-018-9673-y.

[33] Y. Malik, C. R. S. Campos, and F. Jaafar. Detecting android security vulnerabilities using machine learning and system calls analysis. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 109–113, 2019.

[34] S. Afrose, S. Rahaman, and D. Yao. Cryptoapi-bench: A comprehensive benchmark on java cryptographic api misuses. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 49–61, 2019.

[35] J. Gao, L. Li, P. Kong, T. F. Bissyandé, and J. Klein. Understanding the evolution of android app vulnerabilities. *IEEE Transactions on Reliability*, pages 1–19, 2019.

[36] Oliviero Riganelli, Marco Mobilio, Daniela Micucci, and Leonardo Mariani. A benchmark of data loss bugs for android apps. In *Proceedings of the 16th International Conference on Mining Software Repositories*, page 582–586. IEEE Press, 2019. doi: 10.1109/MSR.2019.00087. URL https://doi.org/10.1109/MSR.2019.00087.

[37] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice: Second Edition*. Morgan & Claypool Publishers, 2nd edition, 2017. ISBN 1627057080.

[38] Android. App security best practices, 2020. Available at https://developer.android.com/topic/security/best-practices.

[39] OWASP. Mobile security project, 2019. Available at https://wiki.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Controls.

[40] Carnegie Mellon Software Engineering Institute. Android secure coding standards, 2015. Available at https://wiki.sei.cmu.edu/confluence/display/android/Android+Secure+Coding+Standard.

[41] Matthew Green and Matthew Smith. Developers are not the enemy! the need for usable security apis. *IEEE Security and Privacy*, 14(5):40–46, 2016. ISSN 1540-7993. doi: 10.1109/MSP.2016.111.

[42] CVE. Common vulnerabilities and exposures, 2020. Available at https://cve.mitre.org/.

[43] NIST. National vulnerability database, 2020. Available at https://nvd.nist.gov/.

[44] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 468–471. Association for Computing Machinery, 2016. doi: 10.1145/2901739.2903508.

[45] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. *SIGMETRICS Perform. Eval. Rev.*, 42(1):221–233, 2014. ISSN 0163-5999. doi: 10.1145/2637364.2592003.

[46] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 50–61. Association for Computing Machinery, 2012. ISBN 9781450316514. doi: 10.1145/2382196.2382205.

[47] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[48] Lina Qiu, Yingying Wang, and Julia Rubin. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 176–186. Association for Computing Machinery, 2018. doi: 10.1145/3213846.3213873.

[49] Bradley Reaves, Jasmine Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, Byron Wright, Kevin Butler, William Enck, and Patrick Traynor. *droid: Assessment and evaluation of android application analysis tools. *ACM Comput. Surv.*, 49(3), 2016. doi: 10.1145/2996358.

[50] Joydeep Mitra and Venkatesh-Prasad Ranganath. Ghera: A repository of android app vulnerability benchmarks. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 43–52. ACM, 2017.

[51] S. Calzavara, I. Grishchenko, and M. Maffei. Horndroid: Practical and sound static analysis of android applications by smt solving. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 47–62, 2016.

[52] Felix Pauck, Eric Bodden, and Heike Wehrheim. Do android taint analysis tools keep their promises? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 331–341. ACM, 2018. https://foellix.github.io/ReproDroid/.

[53] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. *SIGPLAN Not.*, pages 169–190, 2006.

[54] C. Isen, L. John, Jung Pil Choi, and Hyo Jung Song. On the representativeness of embedded java benchmarks. In *2008 IEEE International Symposium on Workload Characterization*, pages 153–162, 2008.

[55] James Pallister, Simon J. Hollis, and Jeremy Bennett. Beebs: Open benchmarks for energy measurements on embedded platforms. *CoRR*, abs/1308.5174, 2013.

[56] Jim Gray, editor. *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993. ISBN 1-55860-292-5.

[57] HackerOne. Hackerone – hack for good, 2020. Available at https://www.hackerone.com/.

[58] Google Inc. Shrink code and resources with ProGuard. https://developer.android.com/studio/build/shrink-code, 2017. Accessed: 01-May-2020.

[59] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1025–1035. ACM, 2014.

[60] Alireza Sadeghi, Naeem Esfahani, and Sam Malek. Mining mobile app markets for prioritization of security assessment effort. In *Proceedings of the 2Nd ACM SIGSOFT*

International Workshop on App Market Analytics, WAMA 2017, pages 1–7. ACM, 2017.

[61] Ryan Stevens, Jonathan Ganz, Vladimir Filkov, Premkumar Devanbu, and Hao Chen. Asking for (and about) permissions used by android apps. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 31–40. IEEE Press, 2013.

[62] M. Linares-Vasquez, B. Dit, and D. Poshyvanyk. An exploratory analysis of mobile development issues using stack overflow. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 93–96, 2013.

[63] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering*, 43(6):492–530, June 2017. ISSN 2326-3881. doi: 10.1109/TSE.2016.2615307.

[64] Ashish Bhatia. A collection of android security related resources. https://github.com/ashishb/android-security-awesome, 2014. Accessed: 01-May-2018.

[65] Aditya Agrawal. Mobile Security Wiki. https://mobilesecuritywiki.com/, 2015. Accessed: 01-May-2018.

[66] Nuno Antunes and Marco Vieira. Benchmarking vulnerability detection tools for web services. In *2010 IEEE International Conference on Web Services*, pages 203–210, July 2010. doi: 10.1109/ICWS.2010.76.

[67] Luke Deshotels, Vivek Notani, and Arun Lakhotia. Droidlegacy: Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, pages 3:1–3:12. ACM, 2014.

[68] Feng Shen, Namita Vishnubhotla, Chirag Todarka, Mohit Arora, Babu Dhandapani, Eric John Lehner, Steven Y. Ko, and Lukasz Ziarek. Information flows as a permission

mechanism. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 515–526. ACM, 2014. URL http://doi.acm.org/10.1145/2642937.2643018. http://blueseal.cse.buffalo.edu/multiflow.html, Accessed : 01-June-2018.

[69] Backes SRT GmbH. SRT:AppGuard. http://www.srt-appguard.com/en/, 2014. Accessed: 04-Jun-2018.

[70] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. Practical DIFC enforcement on android. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1119–1136. USENIX Association, 2016. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/nadkarni. https://wspr.csc.ncsu.edu/aquifer/, Accessed : 01-June-2018.

[71] Meng Xu, Chengyu Song, Yang Ji, Ming-Wei Shih, Kangjie Lu, Cong Zheng, Ruian Duan, Yeongjin Jang, Byoungyoung Lee, Chenxiong Qian, and et al. Toward engineering a secure android ecosystem: A survey of existing techniques. *ACM Comput. Surv.*, 49(2), 2016. doi: 10.1145/2963145. URL https://doi.org/10.1145/2963145.

[72] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *22nd USENIX Security Symposium (USENIX Security '13)*. USENIX, 2013. http://www.flaskdroid.org/index.html%3Fpage_id=2.html, Accessed: 04-Jun-2018.

[73] MWR Labs. Drozer. https://github.com/mwrlabs/drozer/, 2012. Accessed: 20-Apr-2018.

[74] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 59:1–59:11. ACM, 2012. URL http://doi.acm.org/10.1145/2393596.2393666.

[75] App-Ray. AppRay. http://app-ray.co/, 2015. Accessed: 04-Jun-2018.

[76] Booz Allen. AppCritique: Online Vulnerability Detection Tool. https://appcritique.boozallen.com/, 2018. Accessed: 21-Nov-2017.

[77] Thomas Debize. AndroWarn : Yet Another Static Code Analyzer for malicious Android applications. https://github.com/maaaaz/androwarn/, 2012. Accessed: 21-Nov-2017.

[78] Adam Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Scandroid: Automated security certification of android applications. 2009. https://github.com/SCanDroid/SCanDroid, Accessed: 04-Jun-2018.

[79] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1092–1104. ACM, 2014. https://www.cs.washington.edu/sparta, Accessed : 01-June-2018.

[80] Yu-Cheng Lin. AndroBugs Framework. https://github.com/AndroBugs/AndroBugs_Framework, 2015. Accessed: 21-Nov-2017.

[81] Amiangshu Bosu, Fang Liu, Danfeng (Daphne) Yao, and Gang Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 71–85, New York, NY, USA, 2017. ACM. https://github.com/dialdroid-android, Accessed: 05-May-2018.

[82] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. ACM, 2014. https://www.cert.org/secure-coding/tools/didfail.cfm, Accessed: 21-Apr-2018.

[83] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In *In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, 2014. https://github.com/utds3lab/SMVHunter, Accessed: 10-Jun-2018.

[84] Cuckoo. CuckooDroid: Automated Android Malware Analysis. https://github.com/idanr1986/cuckoo-droid, 2015. Accessed: 01-May-2018.

[85] Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin" Rinard. Information-flow analysis of Android applications in DroidSafe. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*, 2015. https://github.com/MIT-PAC/droidsafe-src, Accessed: 21-Apr-2018.

[86] Victor van der Veen and Christian Rossow. Tracedroid. http://tracedroid.few.vu.nl/, 2013. Accessed: 01-May-2018.

[87] DevKnox. DevKnox - Security Plugin for Android Studio. https://devknox.io/, 2016. Accessed: 21-Nov-2017.

[88] Duc Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. A stitch in time: Supporting android developers in writingsecure code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1065–1077. ACM, 2017. https://plugins.jetbrains.com/plugin/9497-fixdroid, Accessed: 21-Apr-2018.

[89] Team Keen. Joint Advanced Application Defect Assessment for Android Application (JAADAS). https://github.com/flankerhqd/JAADAS, 2016. Accessed: 21-Nov-2017.

[90] Joaquín Rinaudo and Juan Heguiabehere. Marvin Static Analyzer. https://github.com/programa-stic/Marvin-static-Analyzer, 2016. Accessed: 21-Nov-2017.

[91] Ajin Abraham, Dominik Schelecht, and Matan Dobrushin. Mobile Security Framework. https://github.com/MobSF/Mobile-Security-Framework-MobSF, 2015. Accessed: 21-Nov-2017.

[92] LinkedIn. Quick Android Review Kit. https://github.com/linkedin/qark/, 2015. Accessed: 21-Nov-2017.

[93] David M. W. Powers. Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation. *Journal of Machine Learning Technologies*, 2 (1):37–63, 2011.

[94] IBM. Global Average Total Cost of a Data Breach. https://www.ibm.com/security/data-breach, 2019. Accessed: 01-May-2020.

[95] Leslie Lamport. Who builds a house without drawing blueprints? *Commun. ACM*, 58 (4):38–41, March 2015. ISSN 0001-0782. doi: 10.1145/2736348. URL https://doi.org/10.1145/2736348.

[96] Hillel Wayne. *Practical TLA+: Planning Driven Development*. Apress, USA, 1st edition, 2018. ISBN 1484238281.

[97] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11:256–290, 2002. doi: 10.1145/505145.505149.

[98] Mark Richters and Martin Gogolla. Ocl: Syntax, semantics, and tools. In *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, page 42–68, Berlin, Heidelberg, 2002. Springer-Verlag. ISBN 3540431691.

[99] Jan Jürjens. Umlsec: Extending uml for secure systems development. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *UML 2002 — The Unified Modeling Language*, pages 412–425. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-45800-5.

[100] Henning Heitkotter, Herbert Kuchen, and Tim A. Majchrzak. Extending a model-driven cross-platform development approach for business apps. *Science of Computer Programming*, 97:31 – 36, 2015. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2013.11.013. URL http://www.sciencedirect.com/science/article/pii/S0167642313002979. Special Issue on New Ideas and Emerging Results in Understanding Software.

[101] Marco Brambilla, Andrea Mauri, and Eric Umuhoza. Extending the interaction flow modeling language (ifml) for model driven development of mobile applications front end. In Irfan Awan, Muhammad Younas, Xavier Franch, and Carme Quer, editors, *Mobile Web Information Systems*, pages 176–191. Springer International Publishing, 2014.

[102] Iffat Fatima, Muhammad Waseem Anwar, Farooque Azam, Bilal Maqbool, and Hanny Tufail. Extending interaction flow modeling language (ifml) for android user interface components. In Robertas Damaševičius and Giedrė Vasiljevienė, editors, *Information and Software Technologies*, pages 76–89. Springer International Publishing, 2019.

[103] Zef Hemel and Eelco Visser. Declaratively programming the mobile web with mobl. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, page 695–712, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450309400. doi: 10.1145/2048066.2048121. URL https://doi.org/10.1145/2048066.2048121.

[104] Steffen Vaupel, Gabriele Taentzer, Rene Gerlach, and Michael Guckert. Model-driven development of platform-independent mobile applications supporting role-based app variability. In *Software Engineering 2016*, pages 99–100. Gesellschaft für Informatik e.V., 2016.

[105] M. Usman, M. Z. Iqbal, and M. U. Khan. A model-driven approach to generate mobile applications for multiple platforms. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 111–118, 2014.

[106] Rita Francese, Michele Risi, Giuseppe Scanniello, and Genoveffa Tortora. Model-driven development for multi-platform mobile applications. In *Proceedings of the 16th International Conference on Product-Focused Software Process Improvement - Volume 9459*, page 61–67, Berlin, Heidelberg, 2015. Springer-Verlag. ISBN 9783319268439. doi: 10.1007/978-3-319-26844-6_5.

[107] Xiaoping Jia and Chris Jones. Cross-platform application development using axiom as an agile model-driven approach. In José Cordeiro, Slimane Hammoudi, and Marten van Sinderen, editors, *Software and Data Technologies*, pages 36–51, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-45404-2.

[108] Martin Henkel and Janis Stirna. Pondering on the key functionality of model driven development tools: The case of mendix. In *Perspectives in Business Informatics Research*, pages 146–160. Springer Berlin Heidelberg, 2010.

[109] IBM. Ibm rational rhapsodhy, 2014. Available at `https://www.ibm.com/support/knowledgecenter/SSB2MU_8.2.1/com.ibm.rhp.overview.doc/topics/rhp_c_po_rr_product_overview.html`.

[110] C. Haley, R. Laney, J. Moffett, and B. Nuseibeh. Security requirements engineering: A framework for representation and analysis. *IEEE Transactions on Software Engineering*, 34(1):133–153, 2008.

[111] Torsten Lodderstedt, David Basin, and Jürgen Doser. Secureuml: A uml-based modeling language for model-driven security. In *UML 2002 — The Unified Modeling Language*, pages 36–51, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-45404-2.

[112] Haralambos "Mouratidis and Paolo Giorgini. "secure tropos: A security-oriented extension of the tropos methodology". In *"International Journal of Software Engineering and Knowledge Engineering"*, pages "285–309". "World Scientific", "2007".

[113] Maria Riaz, Jonathan Stallings, Munindar P. Singh, John Slankas, and Laurie Williams. Digs: A framework for discovering goals for security requirements engineering. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, New York, NY, USA, 2016. Association for Computing Machinery. doi: 10.1145/2961111.2962599.

[114] Dmitry Prudnikov. UX design techniques for mobile apps. https://yalantis.com/blog/ux-design-techniques-mobile-app-design/, 2019. Accessed: 05-Feb-2019.

[115] Ambrose Little. Storyboarding in the software design process. https://uxmag.com/articles/storyboarding-in-the-software-design-process, 2013. Accessed: 05-Feb-2019.

[116] Apple. Storyboards in Xcode. https://developer.apple.com/library/archive/documentation/General/Conceptual/Devpedia-CocoaApp/Storyboard.html, 2019. Accessed: 05-Feb-2019.

[117] Sketch. Sketch Design ToolKit. https://www.sketchapp.com/, 2019. Accessed: 05-Feb-2019.

[118] Google Inc. Navigation, 2019. Available at https://developer.android.com/guide/navigation.

[119] Fausto Giunchiglia, John Mylopoulos, and Anna Perini. The tropos software development methodology: Processes, models and diagrams. In Fausto Giunchiglia, James Odell, and Gerhard Weiß, editors, *Agent-Oriented Software Engineering III*, pages 162–173, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-36540-2.

[120] IBM. Android collapses into fragments, 2014. Available at https://securityintelligence.com/wp-content/uploads/2013/12/android-collapses-into-fragments.pdf.

[121] Yu Dongsong. Download arbitrary files to sd card via additional slashes in file: Uri, 2014. Available at https://bugzilla.mozilla.org/show_bug.cgi?id=1050690.

[122] HackerOne. Theft of user session, 2018. Available at https://hackerone.com/reports/328486.

[123] Google Inc. Android lint overview, 2020. Available at http://tools.android.com/lint/overview.

[124] L. Luo, E. Bodden, and J. Spath. A qualitative analysis of android taint-analysis results. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 102–114. IEEE, 2019.

[125] Pascal Gadient, Mohammad Ghafari, Marc-Andrea Tarnutzer, and Oscar Nierstrasz. Web apis in android through the lens of security. In *Proc. SANER*, pages 13–22. IEEE, 2020.

[126] K. Schaffer and J. Voas. What happened to formal methods for security? *IEEE International Conference on Software Maintenance (ICSM)*, 49(8):70–79, 2016.

[127] P. Berander. Using students as subjects in requirements prioritization. In *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE '04.*, pages 167–176, 2004.

[128] Mikael Svahnberg, Aybüke Aurum, and Claes Wohlin. Using students as subjects - an empirical evaluation. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, page 288–290. Association for Computing Machinery, 2008. doi: 10.1145/1414004.1414055.

[129] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. Are students representatives of professionals in software engineering experiments? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, page 666–676. IEEE Press, 2015. ISBN 9781479919345.

[130] Marakana. Android App Development Tutorials. https://www.youtube.com/playlist?list=PLC6F613CB266444FF, 2020. Accessed: 01-May-2020.

Appendix A

Catalog of Benchmarks

In this section, the current benchmarks in Ghera are cataloged according to the vulnerability categories identified in Chapter 2. For each benchmark, I provide a short description of the vulnerability and the exploit that uses the vulnerability.

A.1 Crypto

Crypto APIs help Android apps to encrypt, decrypt information, and manage cryptographic keys.

A.1.1 Block Cipher encryption in ECB mode enables information leak

Vulnerability: Apps that use the Block Cipher algorithm in ECB mode to encrypt sensitive information are vulnerable to information leak.

Exploit: A malicious app on the device with access to the encrypted sensitive information breaks the encryption to get access to the information.

A.1.2 Constant Initialization Vector (IV) enables information leak

Vulnerability: Apps that use the Block Cipher algorithm in CBC mode along with a constant Initialization Vector (IV) to encrypt sensitive information are vulnerable to information leak.

Exploit: A malicious app on the device launch a known plain text attack to guess the IV and break the encryption.

A.1.3 Constant Key saved in app's source code leads to forgery attack

Vulnerability: Apps that use the *Cipher API* need to provide a key. If such a key used for encryption/decryption is saved in the source code, then an attacker can get access to it and abuse it.

Exploit: An attacker obtains the key from from the decompiled source code and uses it in a malicious app to either read sensitive information or inject malicious information.

A.1.4 Keystore without passwords expose cryptographic keys

Vulnerability: Apps can safely store encryption keys in a keystore. But if the keystore is accessible to other apps and the key is not password protected then a malicious app can retrieve it from the keystore. Hence, such apps are susceptible to leaking sensitive information.

Exploit: An attacker accesses the keystore to steal all the keys in the keystore.

A.1.5 Constant Salt for Password Based Encryption leads to information leak

Vulnerability: Apps that use password based encryption to encrypt sensitive information use a salt to generate the password based encryption key. If the salt is not random then the key can be re-generated by a malicious attacker with knowledge of the password.

Exploit: An attacker precomputes a dictionary of symmetric keys for known passwords uses them to decrypt information.

A.2 Inter Component Communication

Android apps are composed of four basic kinds of components: 1) *Activity* components display the user interface, 2) *Service* components perform background operations, 3) *Broadcast Receiver* components receive event notifications and act on those notifications, and 4) *Content Provider* components manage app data. Communication between components in an app and in different apps is facilitated via exchange of *Intent*s. Components specify their ability to process specific kinds of intents by using *intent-filters*.

A.2.1 Dynamically registered broadcast receiver provides unrestricted access

Vulnerability: When a broadcast receiver is dynamically registered with the Android platform, a non-null intent filter is provided. As a result, the component is automatically exported to be accessible from other apps, including malicious apps.

Exploit: A malicious app broadcasts a message to a dynamically registered broadcast receiver. This triggers the broadcast receiver to process the intent and unintentionally perform an action on behalf of the malicious app.

A.2.2 Empty pending intent leaks privilege

Vulnerability: An app X can allow another app Y to perform an action on its behalf at a future time via a *pending intent*; these intents are saved in the system. When no action is specified in a pending intent, the recipient of the pending intent can set any action and execute it in the context of the app that sent the pending intent.

Exploit: A malicious app specifies its interest in the pending intent via an intent-filter. Upon receiving an empty pending intent, the malicious app associates a malicious action with the pending intent. Consequently, when the pending intent is processed, the malicious action will be executed in the context of app X.

A.2.3 Unverified fragment loading enables privilege escalation

Vulnerability: A fragment is a reusable class implementing a portion of an activity. An activity can accept a fragment name as input and load it at runtime. If the fragment name is not validated then any fragment can be loaded in an activity.

Exploit: A malicious app on the device uses the benign app to load a fragment of its choice at runtime.

Vulnerability: Android platform uses intent-filters to identify the service to process *implicit intents*, that is, intents dedicated to a class of targets (as opposed to specific target). When multiple services have the same intent-filter, the service with higher priority is chosen to process corresponding intents.

Exploit: A malicious app has a service X with the same intent-filter as that of the service Y in a benign app and with higher priority than Y. When an app requests the start of service Y by relying on the intent-filter, service X in the malicious app will be started.

A.2.4 Low priority activity prone to hijacking

Vulnerability: A *priority* can be specified for an activity in the app's manifest file. When an activity is started, Android displays all activities with the same intent-filter as a list to the user in the order of priority (high to low).

Exploit: A malicious app registers an activity X with the same *intent-filter* as that of an activity Y registered by a benign app and with higher priority than Y. Consequently, the malicious app's activity X will be displayed before the benign app's activity Y.

A.2.5 Implicit pending intent leaks information

Vulnerability: A app X can create a pending intent containing an implicit intent. When the pending intent is processed, the containing implicit intent will be processed by a component identified based on the intent-filter. When multiple components have the same intent-filter, the component with higher priority is chosen to process corresponding intents.

Exploit: A malicious app has a component X with an intent-filter same as that of the component Y in the benign app and X has higher priority than Y. So, component X is chosen (over component Y) to process the implicit intent in the pending intent.

A.2.6 Content provider with inadequate path-permission leaks information

Vulnerability: An app can use *path-permissions* to control access to the data exposed by a content provider. When an app protects a folder by permissions, only the files in the folder are protected by the permissions; none of the subfolders and their descendants are protected by the permissions.

Exploit: A malicious app calls methods of a content provider to access and modify subdirectories and contained files that are not protected by path-permissions.

A.2.7 Incorrectly handling implicit intent leads to unauthorized access

Vulnerability: A component that deals with sensitive information or performs sensitive operations should be careful when handling implicit intents because accepting implicit intents *exports* the component.

Exploit: A malicious app can access such a component with a crafted intent to steal sensitive information or perform unauthorized operations.

A.2.8 Apps have unrestricted access to Broadcast receivers registered for system events

Vulnerability: When a Broadcast receiver registers to receive (system) intents from the Android platform, it needs to be exported. Consequently, it is accessible by any app without restrictions.

Exploit: A malicious app sends an intent to a broadcast receiver that is registered to receive system intents and possibly forces it to perform unintended operations.

A.2.9 Ordered broadcasts allow malicious data injection

Vulnerability: When an ordered broadcast is sent, broadcast receivers respond to it in the order of priority. Broadcast receivers with higher priority respond first and forward it to receivers with lower priority.

Exploit: A malicious receiver with high priority receives the intent, changes it, and forwards it to lower priority receivers.

A.2.10 Sticky broadcasts are prone to leaking sensitive information and malicious data injection

Vulnerability: When a *sticky broadcast message (intent)* is sent, it is delivered to every registered receiver and is saved in the system to be provided to receivers that register for the message in the future. When the message is re-broadcasted with modification, the modified message replaces the original message in the system.

Exploit: A malicious broadcast receiver registers for the message at later time and retrieves any sensitive information in the message. Further, it can modify the contents of the message and re-broadcast to provide incorrect information to future receivers of the message.

A.2.11 Non-empty task affinity of a non-launcher activity makes an app vulnerable to phishing attacks via back button

Vulnerability: A *task* is a collection (stack) of activities. When an activity is started, it is launched in a task. An activity can request that it be started in a specific task. This is known as *task affinity*. If the back button is pressed from an activity 'A' then the activity below 'A' in the task stack will be displayed as opposed the activity that started 'A'.

Exploit: An activity X in a malicious app has the same task affinity as an activity Y in a benign app. If activity X is started before activity Y then pressing the back button from activity Y will bring activity X to the foreground.

A.2.12 Non-empty task affinity of a non-launcher activity makes an app vulnerable to phishing attacks

Vulnerability: A *task* is a collection (stack) of activities. When an activity is started, it is launched in a task. An activity can request that it be started in a specific task. This is known as *task affinity*. The task containing the displayed activity is moved to the background if none of the activities in that task are being displayed. When any activity from a task in the background is resumed, then the activity at the top of the task (and not the resumed activity) is displayed.

Exploit: An activity X in a malicious app requests to start itself in the same task as an activity Y in a benign app. When activity X is at the top of the task, any call to activity Y will cause activity X to be displayed to the user.

A.2.13 Non-empty Task affinity of a launcher activity makes an app vulnerable to phishing attacks

Vulnerability: Android starts every activity in a task, based on task affinity[1]. If the activity being started is a *launcher activity* then Android requires that the *launcher activity* be the first activity in the task.

Exploit: An activity M1 in a malicious app requests to start itself in the same task as B1, *launcher activity* in a benign app. If activity M1 is started before B1 then Android will refuse to start B1 even if the user explicitly requests Android to start B1.

A.2.14 Task affinity and task re-parenting enables phishing and denial-of-service

Vulnerability: An activity can request to always be at the top of a task. This is called *task re-parenting.* In such cases, when an activity from that task resumed, activity at the top of the task will be displayed to the user.

Exploit: An activity in a malicious app uses task affinity and task re-parenting to supersede activities from other apps in a task and launch a denial-of-service attack or a phishing attack.

[1]The default task affinity of an activity is the package name of the app

A.2.15 Unhandled exceptions enable Denial-Of-Service

Vulnerability: Apps that fail to handle exceptions while servicing incoming intents can be crashed by sending appropriately crafted intents. Hence, such apps are vulnerable to Denial of Service attack.

Exploit: A malicious app can inject a null value into an intent for a benign app. When the benign app acts on the intent, it crashes.

A.2.16 Unprotected components allow unexpected action without user approval

Vulnerability: If an app has the permission to perform a particular privileged operation and if that operation is performed via an app component that is exported for public consumption, then a malicious app can invoke the component method that performs the privileged operation and make the app perform the operation on the behalf of the malicious app.

Exploit: Malicious invokes an exported broadcast receiver in *Benign* to send SMS without having the permission to send SMS.

A.2.17 Content Provider API allow unauthorized access

Vulnerability: Content provider API provides a method `call` to call any provider-defined method. With a reference to the content provider, this method can be invoked without any restrictions.

Exploit: A malicious app uses `call` method to invoke content provider methods to access the underlying data even when it does not have specific permissions to access this data.

A.3 Networking

Networking APIs allow Android apps to communicate over the network via multiple protocols.

A.3.1 Invalid remote server certificate enables MITM

Vulnerability: Android apps can use SSL/TLS to securely communicate with a web server via a chain of certificates signed by trusted CAs. Every certificate in the certificate chain has an expiration date which can be checked by the app. If an app connects to a server without checking the expiration date of a certificate then the app might connect to a malicious server using the expired certificate.

Exploit: A malicious server can use a certificate with an expired date and masquerade as the legitimate server. Hence, the app will connect with the malicious application instead of the legitimate server.

A.3.2 Incorrect server host name enables MITM

Vulnerability: Apps can employ HostnameVerifier interface to perform custom checks on host name when using SSL/TLS for secure communication. If these checks are incorrect, apps can end up connecting to malicious servers and be targets of malicious actions.

Exploit: A malicious server can use a host name to bypass the weak/incorrect host name verification and connect with the app.

A.3.3 The InsecureSSLSocket API enables MITM

Vulnerability: Apps that use the `SSLCertificateSocketFactory.getSocket(InetAddress, ...)` method are vulnerable to MITM attacks.

Exploit : A malicious server can masquerade as a legitimate server and connect with the app.

A.3.4 The InsecureSSLSocketFactory API enables MITM

Vulnerability: Apps that use the SSLCertificateSocketFactory.getSocket(InetAddress, ...) method are vulnerable to MITM attacks.

Exploit: A malicious server can masquerade as a legitimate server and connect with the app.

A.3.5 Invalid Certificate Authority enables MITM

Vulnerability: In secure communication, apps employ TrustManager interface to check the validity and trustworthiness of presented certificates. If these checks are incorrect, apps can end up trusting certificates from malicious servers and be targets of malicious actions.

Exploit: A malicious server can use a certificate signed by an invalid certificate authority to masquerade as the legitimate server and connect with the app.

A.3.6 Writing to open socket enables information leak

Vulnerability: Apps that send information to a remote server over an open socket are vulnerable to information leak.

Exploit: A malicious app on the device connects to the open port and reads sensitive information.

A.3.7 Un-encrypted socket communication enables IP spoofing attacks

Vulnerability: Apps that communicate with a server over TCP/IP without encryption allow Man-in-the-Middle attackers to spoof server IPs by intercepting client-server data streams.

Exploit: A malicious application takes advantage of lack of encryption and can launch an IP spoofing attack.

A.3.8 Unpinned Certificates enables MITM

Vulnerability: An app can store a certificate it trusts and only trust that certificate when connecting to a web server. This is called certificate pinning. Not pinning a trusted certificate can compromise an app's security if a certificate trusted by the device is compromised.

Exploit : A malicious server can use a certificate signed by a certificate authority trusted by Android, masquerade as a legitimate server, and connect with the app.

A.4 NonAPI

Android apps use third party libraries to use features that do not come packaged with the underlying Android framework APIs.

A.4.1 Apps that use libraries with vulnerable manifest are vulnerable

Vulnerability: When apps use libraries, the manifest file of the library is merged with the manifest file of the app. If the manifest of the library has a vulnerability e.g., exported component that should not be exported, then the app unknowingly inherits the vulnerability.

Exploit: A malicious on the device can trigger a component in the third-party library without the necessary permissions.

A.4.2 Apps that use outdated libraries might be vulnerable

Vulnerability: When an app uses an external library, it may use older versions of the library which may contain vulnerabilities. As a result, the app may exhibit vulnerabilities as result of using an outdated library.

Exploit: A malicious application may exploit the app by exploiting the vulnerabilities in the outdated library.

A.5 Permission

Android apps run in a sandbox. Hence, they need permissions to use features outside the sandbox.

A.5.1 Unnecessary permissions enable privilege escalation

Vulnerability: Android requires apps to explicitly request for permission when invoking a protected API. However, if an app asks for more permissions than necessary then the permission can be (mis)used by less privileged apps to invoke protected APIs.

Exploit: A malicious app on the device can use a protected API via an app that permissions to access the protected API.

A.5.2 *Normal* Permissions are granted to apps at install time

Vulnerability: If an exported component is protected with a *normal* permission then any app (including a malicious app), that specifies that it needs to use this permission will be granted this permission by Android when the app is installed into the device.

Exploit: A malicious app requesting permission P, if installed on the device will be able to access a component protected with permission P without explicitly asking the user.

A.6 Storage

Android provides numerous options for storing application data. It provides

1. *Internal Storage* to store data that is private to apps. Every time an application is uninstalled, its internal storage is emptied. Starting from Android 7.0, files stored in internal storage cannot be shared with other apps.

2. *External Storage* as a data storage area that is common to apps. Its public partition is accessible to any app without any restrictions. Its private partition is only accessible to apps with a specific permission.

A.6.1 External storage allows data injection attack

Vulnerability: Files stored in external storage can be modified by an app with (appropriate) access to external storage.

Exploit: A malicious app modifies external storage (e.g., add files) and the content in external storage (e.g., change files).

A.6.2 Writing sensitive information to external storage enables information leak

Vulnerability: Files stored in external storage can be accessed by an app with (appropriate) access to external storage.

Exploit: A malicious app reads content from external storage.

A.6.3 Accepting file path as input from external sources leads to directory traversal attack

Vulnerability: If an app accepts a file path as input from an external source like the user, or another app, or the web, and does not sanitize the input then it is possible to craft a malicious input to obtain access to the app's internal file-system and read, write, execute files in it.

Exploit: A malicious app sends an intent to the vulnerable app with a crafted path embedded in the intent.

A.6.4 Writing sensitive information in internal storage to external storage enables information leak

Vulnerability: If an app has a component that takes internal file path as input from untrusted sources and writes writes its contents to external storage then the app is leaking information.

Exploit: A malicious app can craft a file path to download sensitive files in internal storage to external storage. From external storage it can then read the downloaded files.

A.6.5 The execSQL() API is vulnerable to SQL injection attacks

Vulnerability: SQLiteDatabase.execSQL() method can be used by apps to update data. If such uses rely on external inputs and use non-parameterized SQL queries, then they are susceptible to sql injection attack.

Exploit: A malicious app creates a query and sends it to execSQL in the benign app, which executes it.

A.6.6 The rawQuery() API is vulnerable to SQL injection attacks

Vulnerability: Apps that use SQLiteDatabase.rawQuery() method to construct non-parameterized SQL queries are vulnerable to SQL injection attacks.

Exploit: A malicious app creates a specially crafted query to retrieve sensitive information from the vulnerable app that uses *rawQuery*.

A.6.7 The absence of *selectionArgs* in SQLite queries can result in SQL injection attacks

Vulnerability: Apps that do not use *selectionArgs* to construct SQLite queries are vulnerable to SQL injection attacks.

Exploit: A malicious app constructs a crafted query to retrieve all sensitive information from the vulnerable app's SQLite database.

A.7 System

System APIs help Android apps access low level features of the Android platform like process management, thread management, runtime permissions etc.

Every Android app runs in its own process with a unique Process ID (PID) and a User ID (UID). All components in an app run in the same process. A permission can be granted to an app at installation time or at run time. If an app is granted a specific permission at installation time, then all components of the app are granted the same permission. If component in an app is protected by a permission, only components that have been granted this permission can communicate with the protected component. If the permission is checked at runtime, then all components have to request for the required permission.

A.7.1 checkCallingOrSelfPermission method leaks privilege

Vulnerability: Before servicing a request, a component protected by a permission uses `checkCallingOrSelfPermission` to check if the requesting component has the permission. This method returns true if the app containing the requesting component or the app containing the protected component has the given permission. When the app containing the protected component has the permission, the method will always return true.

Exploit: A malicious app accesses a component that is protected by permission P, is in an app that has permission P, and uses `checkCallingOrSelfPermission` to check for permission.

A.7.2 checkPermission method leaks privilege

Vulnerability: Before servicing a request, a component protected by a permission uses `checkPermission` to check if the given PID and UID pair has the permission. Typically, `getCallingPID` and `getCallingUID` methods of Binder API are used to retrieve PID and UID, respectively. When these methods are invoked in the main thread of an app, they return the IDs of the app and not the IDs of the calling app.

Exploit: A malicious app accesses a component that is protected by permission P, is in an app that has permission P, and uses `checkPermission` to check for permission in the main thread of the containing app.

A.7.3 Writing sensitive information to clipboard enables information leak

Vulnerability: Android allows all apps to access clipboard information. If an app allows users to copy sensitive information then that information is written to Clipboard. A malicious app can access Clipboard and retrieve the sensitive information.

Exploit: A malicious app on the device reads data from the clipboard.

A.7.4 Unverified code loading enables code injection attacks

Vulnerability: Android apps can use dynamic code loading features to dynamically load and execute code not packaged with the app. If the app does not verify the integrity and authenticity of the code before dynamically loading and executing it then the app is vulnerable to code injection attacks.

Exploit: A malicious application injects code, which when loaded will execute the context if the benign app.

A.7.5 enforceCallingOrSelfPermission method leaks privilege

Vulnerability: Before servicing a request, a component protected by a permission uses `enforceCallingOrSelfPermission` to check if the requesting component has the permission. This method raises `SecurityException` if the app containing the requesting component or the app containing the protected component does not have the given permission. When the app containing the protected component has the permission, the method will complete without any exceptions.

Exploit: A malicious app accesses a component that is protected by permission P, is in an app that has permission P, and uses `enforceCallingOrSelfPermission` to enforce the permission.

A.7.6 enforcePermission method leaks privilege

Vulnerability: Before servicing a request, a component protected by a permission uses `enforcePermission` to check if the given PID and UID pair has the permission. Typically, `getCallingPID` and `getCallingUID` methods of Binder API are used to retrieve PID and

UID, respectively. When these methods are invoked in the main thread of an app, they return the IDs of the app and not the IDs of the calling app.

Exploit: A malicious app accesses a component that is protected by permission P, is in an app that has permission P, and uses `enforcePermission` to enforce the permission in the main thread of the containing app.

A.7.7 Collecting device identifier may enable information exposure

Vulnerability: An app can use device identifiers to identify unique app instances. Collecting such information could lead to exposing sensitive data related to the device. Malicious actors can use such information to track the device.

Exploit: A malicious application accesses device ID, if the device ID is stored in a publicly accessible location.

A.8 Web

Web APIs allow Android apps to interact with web servers both insecurely and securely (via SSL/TLS), display web content through *WebView* widget, and control navigation between web pages via *WebViewClient* class.

A.8.1 Connecting via HTTP enables Man-in-the-Middle (MitM) attack

Vulnerability: Android apps that use HTTP to connect to remote servers are vulnerable to information theft and IP spoofing attacks.

Exploit: An application takes advantage of the lack of a secure connection and mounts a MitM attack.

A.8.2 Allowing execution of unverified JavaScript code in WebView exposes app's resources

Vulnerability: When an app uses `WebView` to display web content and any JavaScript code embedded in the web content is executed, the code is executed with the same permission as the `WebView` instance used in the app.

Exploit: An app injects malicious JavaScript code into the web content loaded in `WebView` (e.g., modify static web page stored on the device).

A.8.3 Allowing cookie access in WebViews enables cookie overwrite attacks

Vulnerability: Apps that allow websites viewed through `WebView` may enable cookie overwrite attacks.

Exploit: A malicious application overwrites the cookies in the webpage displayed via `WebView`.

A.8.4 Unsafe Handling of Intent URIs leads to information leak

Vulnerability: Apps that do not safely handle an incoming intent embedded inside a URI are vulnerable to information leak via intent hijacking.

Exploit: An application takes advantage of lack of validation on intent URIs and embeds a malicious intent in a web page.

A.8.5 Absence of explicit user approval leads to unintended information exposure

Vulnerability: Apps that disclose sensitive information without explicitly requesting the user for permission are vulnerable to unintended information exposure

Exploit: An application takes advantage of lack of user approval and exploits the vulnerable app to disclose sensitive information.

A.8.6 Allowing unverified JavaScript code enables unauthorized access to an app's content providers

Vulnerability: Apps that allow Javascript code to execute in a WebView without verifying where the JavaScript is coming from, can expose the app's resources.

Exploit: A malicious app injects JavaScript code into the `WebView` to access an app's content providers without the necessary permission.

A.8.7 Allowing unverified JavaScript code enables unauthorized access to an app's files

Vulnerability: When WebView is used to display web content, JavaScript code executed as part of the web content is executed with the permissions of the host app. Without proper checks, malicious JavaScript code can get access to the app's private files.

Exploit: A malicious app injects JavaScript code into the `WebView` to access an app's files without restriction.

A.8.8 Ignoring SSL errors in WebViewClient enables Man-in-the-Middle (MitM) attack

Vulnerability: When an app loads web content from a SSL connection via `WebView` and is notified of an SSL error while loading the content (via `onReceivedSslError` method of `WebViewClient`), the app ignores the error.

Exploit: An application takes advantage of ignored errors and mounts a MitM attack.

A.8.9 Lack of validation of resource load requests in WebView allows loading malicious content

Vulnerability: When a resource (e.g., CSS file, JavaScript file) is loaded in a web page in `WebView`, the app does not validate the resource load request in `shouldInterceptRequest` method of `WebViewClient`. Consequently, any resource will be loaded into `WebView`.

Exploit: An application takes advantage of lack of validation of resource load requests and mounts a MitM attack.

A.8.10 Web content with file scheme base URL enables unauthorized access of an app's resources

Vulnerability: Apps that use loadDataWithBaseUrl() with a file scheme based baseURL (e.g., file://www.google.com) and allow the execution of JavaScript sourced from unverified source may expose the app's resources.

Exploit: A malicious application inject JavaScript to access files in the benign app.

A.8.11 Lack of validation web page load requests in WebView allows loading malicious content

Vulnerability: When a web page is to be loaded into `WebView`, the app does not validate the web page load request in `shouldOverridUrlLoading` method of `WebViewClient`. Consequently, any web page provided by the server will be loaded into `WebView`.

Exploit: An application takes advantage of lack of validation of web page load requests and mounts a MitM attack.

A.8.12 The HttpAuthHandler#proceed API enables unauthorized access to a web service/resource

Vulnerability: Apps that use `HttpAuthHandler#proceed(username, password)` to instruct the WebView to perform authentication with the given credentials may give unauthorized access to third-party when the apps do not validate credentials, e.g., by sending token of previously validated credentials.

Exploit: A malicious app uses previously validated credentials to bypass the authentication.