University of California
Santa Barbara

Identifying and Mitigating Trust Violations in the Mobile Ecosystem

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Antonio Bianchi

Committee in charge:

Professor Giovanni Vigna, Co-Chair
Professor Christopher Kruegel, Co-Chair
Professor Tevfik Bultan

September 2018

The dissertation of Antonio Bianchi is approved.

Professor Tevfik Bultan

Professor Christopher Kruegel, Co-Chair

Professor Giovanni Vigna, Co-Chair

July 2018

Acknowledgements

First of all, I want to thank my family, because they have always helped me, in their own way.

Surely, I have to thank Yanick Fratantonio, because you are *the reason why* I ended up at UCSB, and Jacopo Corbetta, because you are a wise friend and *the only one that really knows how things work*. Also, I want to thank Ruoyu "Fish" Wang, because you are *a good genius*, and Yan Shoshitaishvili, because you are *a good leader*. Moreover, I cannot avoid saying thank you to Aravind Machiry, because you are *a nofos* person, and Eric Gustafson, because you are *an always positive* person.

Obviously, I also need to say a huge thank you to my advisors, Professor Giovanni Vigna and Professor Christopher Kruegel. You not only guided me through my Ph.D., but you also gave me the opportunity to meet great persons, visit interesting places, and end up in "very strange" situations. Your example is something that (shamelessly quoting Yan Shoshitaishvili's thesis) *"I will strive to emulate throughout my academic career, and it is my hope that a student of mine will one day be as thankful to me as I am to them."*

All the people I have worked with at SecLab own my respect and gratitude. I cannot mention you all, but you know who you are.

Finally, to all the great teachers, professors, colleges, and friends I had the honor to meet during my career as a student, I want to say that your passion has been my strongest motivation.

Curriculum Vitæ
Antonio Bianchi

Education

2012 – 2018	Ph.D. in Computer Science Security Lab — Computer Science Department University of California, Santa Barbara. *GPA: 4.0 out of 4.0*
2009 – 2012	M.Sc. in Computer Science University of Illinois at Chicago. *GPA: 3.71 out of 4.0*
2008 – 2012	M.Sc. in Computer Engineering Politecnico di Milano, Italy. *Final grade: 110 cum laude out of 110*
2005 – 2008	B.Sc. in Computer Engineering Politecnico di Milano, Italy. *Final grade: 108 out of 110*

Research and Professional Experience

2012 – 2018	Research Assistant Security Lab — Computer Science Department University of California, Santa Barbara
2017	Research Intern Institute for Information Security & Privacy Georgia Institute of Technology
2011	Visiting Researcher Security Lab — Computer Science Department University of California, Santa Barbara
2010 – 2011	Student Tutor Politecnico di Milano, Italy
2006 – 2008	Web Developer

Publications

1. Yan Shoshitaishvili, **Antonio Bianchi**, Kevin Borgolte, Amat Cama, Jacopo Corbetta, Francesco Disperati, Audrey Dutcher, John Grosen, Paul Grosen, Aravind Machiry, Chris Salls, Nick Stephens, Ruoyu Wang, Giovanni Vigna.
   Mechanical Phish: Resilient Autonomous Hacking.
   In IEEE Security & Privacy Magazine – SPSI: Hacking without Humans, Mar 2018.

2. **Antonio Bianchi**, Yanick Fratantonio, Aravind Machiry, Christopher Kruegel, Giovanni Vigna, Simon Pak Ho Chung, Wenke Lee.
   Broken Fingers: On the Usage of the Fingerprint API in Android.
   In Proceedings of the Network & Distributed System Security Symposium (NDSS), Feb 2018.

3. **Antonio Bianchi**, Eric Gustafson, Yanick Fratantonio, Christopher Kruegel, Giovanni Vigna.
   Exploitation and Mitigation of Authentication Schemes Based on Device-Public Information.
   In Proceedings of the Annual Computer Security Applications Conference (ACSAC), Dec 2017.

4. Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, **Antonio Bianchi**, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna.
   BootStomp: On the Security of Bootloaders in Mobile Devices.
   In Proceedings of the USENIX Security Symposium (Usenix SEC), Aug 2017.

5. Aravind Machiry, Eric Gustafson, Chad Spensky, Chris Salls, Nick Stephens, Ruoyu Wang, **Antonio Bianchi**, Yung Ryn Choe, Christopher Kruegel, Giovanni Vigna.
   BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments.
   In Proceedings of the Network & Distributed System Security Symposium (NDSS), Feb 2017.

6. Ruoyu Wang, Yan Shoshitaishvili, **Antonio Bianchi**, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, Giovanni Vigna.
   Ramblr: Making Reassembly Great Again.
   In Proceedings of the Network & Distributed System Security Symposium (NDSS), Feb 2017. ***Distinguished Paper Award***

7. **Antonio Bianchi**, Kevin Borgolte, Jacopo Corbetta, Francesco Disperati, Andrew Dutcher, John Grosen, Paul Grosen, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Nick Stephens, Giovanni Vigna, Ruoyu Wang — *Authors listed alphabetically.*
   Cyber Grand Shellphish.
   In Phrack Magazine, Jan 2017.

8. Yanick Fratantonio, **Antonio Bianchi**, William Robertson, Engin Kirda, Christopher Kruegel, Giovanni Vigna.
   TriggerScope: Towards Detecting Logic Bombs in Android Apps.
   In Proceedings of the IEEE Symposium on Security and Privacy (S&P), May 2016.

9. Vitor Afonso, **Antonio Bianchi**, Yanick Fratantonio, Adam Doup, Mario Polino, Paulo de Geus, Christopher Kruegel, Giovanni Vigna.
   Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy.
   In Proceedings of the Network & Distributed System Security Symposium (NDSS), Feb 2016.

10. Simone Mutti, Yanick Fratantonio, **Antonio Bianchi**, Luca Invernizzi, Jacopo Corbetta, Dhilung Kirat, Christopher Kruegel, Giovanni Vigna.
    BareDroid: Large-Scale Analysis of Android Apps on Real Devices.
    In Proceedings of the Annual Computer Security Applications Conference (ACSAC), Dec 2015.

11. **Antonio Bianchi**, Yanick Fratantonio, Christopher Kruegel, Giovanni Vigna.
NJAS: Sandboxing Unmodified Applications in non-rooted Devices Running Stock Android.
In Proceedings of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM), Oct 2015.

12. Yanick Fratantonio, Aravind Machiry, **Antonio Bianchi**, Christopher Kruegel, Giovanni Vigna.
CLAPP: Characterizing Loops in Android Applications.
In Proceedings of the Symposium on the Foundations of Software Engineering (FSE), Sep 2015.

13. Yanick Fratantonio, Aravind Machiry, **Antonio Bianchi**, Christopher Kruegel, Giovanni Vigna.
CLAPP: Characterizing Loops in Android Applications.
In Proceedings of International Workshop on Software Development Lifecycle for Mobile (DeMobile), Aug 2015.

14. Yanick Fratantonio, **Antonio Bianchi**, William Robertson, Manuel Egele, Christopher Kruegel, Engin Kirda, Giovanni Vigna.
On the Security and Engineering Implications of Finer-Grained Access Controls for Android Developers and Users.
In Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), Jul 2015.

15. **Antonio Bianchi**, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, Giovanni Vigna.
What the App is That? Deception and Countermeasures in the Android User Interface.
In Proceedings of the IEEE Symposium on Security and Privacy (S&P), May 2015.

16. Yinzhi Cao, Yanick Fratantonio, **Antonio Bianchi**, Manuel Egele, Christopher Kruegel, Giovanni Vigna, Yan Chen.
EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework.
In Proceedings of the Network & Distributed System Security Symposium (NDSS), Feb 2015.

17. Yinzhi Cao, Yanick Fratantonio, **Antonio Bianchi**, Manuel Egele, Christopher Kruegel, Giovanni Vigna, Yan Chen.
Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications.
In Proceedings of the Network & Distributed System Security Symposium (NDSS), Feb 2014.

18. **Antonio Bianchi**, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna.
Blacksheep: Detecting Compromised Hosts in Homogeneous Crowds.
In Proceedings of the ACM Conference on Computer and Communications Security (CCS), Oct 2012.

Abstract

Identifying and Mitigating Trust Violations
in the Mobile Ecosystem

by

Antonio Bianchi

Mobile systems, such as smartphones and tablets, are now the most common way users handle digital information and interact with online services. The interaction with these devices encompasses different actors, trusting each other in different ways. Users interact with apps, trusting them to access valuable and privacy-sensitive information. At the same time, apps usually communicate with remote backends and mediate user authentication to online services. Finally, all these interactions are mediated, on one side, by the user interface and, on the other, by the operating system.

In this thesis, I will present my studies on how all these different actors trust each other and how this trust can be unfortunately violated by attackers. This is possible due to limitations on how the operating system, apps, and the user interface are currently designed and implemented. To assist my work, I developed automatic analysis tools to perform large-scale analyses of Android apps. In this thesis, I will describe both the tools I have developed and my findings.

Specifically, I will first describe my work on how, in an Android system, it is possible to lure users to interact with malicious apps which "look like" legitimate ones. This completely violates the trust relationship, mediated by the user interface, between users and apps. As a countermeasure, I implemented modifications of the Android user interface and evaluated their effectiveness with a user study. Then, I will explain how many apps unsafely authenticate their users to remote backends, due to misplaced trust in

the operating system. In particular, I identified different apps that only rely on values provided by the operating system to perform authentication. For this reason, an attacker can trivially spoof these values, and logins in behalf of a legitimate user. Finally, I will show how many apps misuse hardware-backed authentication devices, such as trusted execution environments and fingerprint readers, making them vulnerable to a variety of authentication bypass attacks

Contents

Chapter 1

Introduction

Mobile systems, such as smartphones and tablets, have become part of the everyday life of billions of people. Given the ability of these devices to continuously collect potentially-private information from the environment around them, and the fact that users trust them to perform an increasing amount of sensitive tasks, in an ideal world, the software they run should be free from security issues. However, the complexity of these devices and of the ecosystem revolving around them (which is composed of millions of apps, application markets, different manufacturers, and multiple vendors) hinders the developers' ability to safely implement functionalities. The goal of my research has been to study the new security issues introduced by the mobile ecosystem, showing how they can be exploited by attackers and implementing solutions for these problems.

As an introductory example, consider a user using an app on a mobile device to transfer money. This simple and common use case hides several challenges that my research has unveiled and addressed: How can the user be sure that they are interacting with the legitimate mobile banking app (and not with a look-alike malicious application)? How can the app authenticate the user to the remote banking backend? Is password-

based authentication sufficient? What are the alternatives to improve both usability and security? Can authentication and authorization be implemented securely, even when a device is in a non-secure state (e.g., the operating system is compromised)?

All these questions correspond to currently open problems in the field of *Mobile System Security*. As a result, smartphones are exploited every day by attackers, causing privacy violations, monetary loss, and even property loss when, for instance, a smartphone is used to control a house's lock.

In this section, first I will introduce the lines of research I followed and my general approach. Then, I will briefly present the three projects that form the main body of this thesis.

1.1 Lines of Research and Approach

In this thesis work, I followed two complementary lines of research:

- **Understanding the trust relationships in the mobile ecosystem:** Mobile systems involve different actors, such as users, user-interfaces, apps, devices, hardware, remote backends, and hardware manufacturers. Understanding how they interact is crucial to evaluate the security of a mobile system.

- **Developing scalable automated analyses:** To evaluate the security of mobile systems *rigorously* and *at scale*, a combination of static and dynamic automated analyses is necessary. Existing techniques need to be adapted to work in the mobile ecosystem.

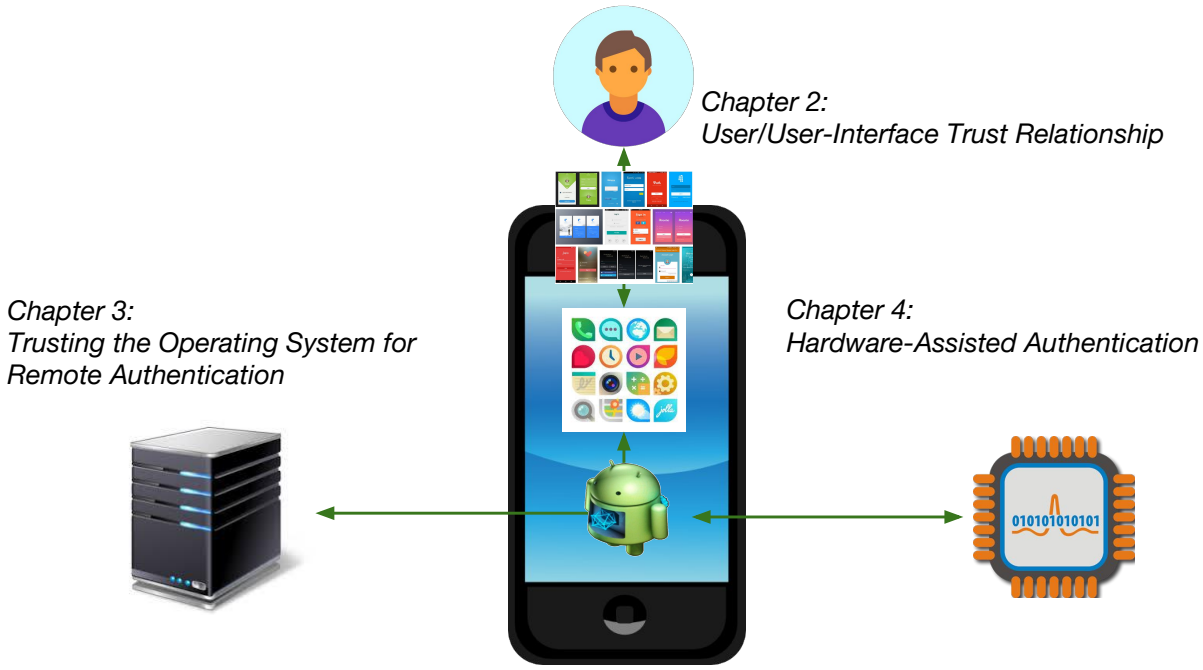


Figure 1.1: A summary of the different mobile ecosystem's components studied in this thesis.

1.1.1 Understanding the Trust Relationships in the Mobile Ecosystem

The interaction with mobile systems encompasses different actors, trusting each other in different ways. First of all, we have users, interacting with apps and trusting them to access valuable and privacy-sensitive information. At the same time, apps usually communicate with remote backends and mediate user authentication to online services. Finally, all these interactions are mediated on one side by the user interface and on the other by the operating system.

If any of these components is compromised the entire "chain-of-trust" is violated. For instance, consider the case in which a user wants to use a mobile banking app. If an attacker can compromise the device's user interface (for instance, luring the user to insert her credentials in an attacker-controlled app, instead of the legitimate one) the

entire authentication schema is defeated, regardless of how well its other components are implemented. Likewise, if the application stores an authentication token in a publicly readable location, the authentication schema can be violated by an attacker able to install a malicious app on the victim's device.

Figure 1.1 presents an overview of the different components in the mobile ecosystem studied by this thesis and their relationships. On the top, we have users interacting with apps. The user interface mediates this interaction. How this interaction is carried out and how attackers can compromise it will be the topic of Chapter 2. On the left, we have remote backends. In fact, most of the apps continuously communicate with remote backends and implement some form of authentication between their users and their remote backends. The way in which an attacker can exploit some of the authentication schemas used by these apps, which are vulnerable because they use untrusted information to perform authentication, will be the topic of Chapter 3. Finally, on the right, the figure shows Trusted Execution Environments (TEEs), which, in modern devices, can be used to implement safer and more usable authentication schemas. The way in which apps use (often incorrectly) TEEs and weaknesses in their current design will be the topic of Chapter 4.

1.1.2 Developing Scalable Automated Analyses

The sheer amount of applications available for mobile systems and the complexity of these applications rule out manual analysis to carry on any comprehensive security analysis. For this reason, throughout my research I have developed tools, based both on static and dynamic analysis, to *automatically* study security issues *at scale*.

These tools are not only useful for security researchers, but also for developers to identify vulnerabilities in their apps, and for application market operators to vet submitted applications for both vulnerabilities and malicious code. In fact, automated analysis of mobile apps is in itself an open research problem, since the mobile environment poses significant challenges to this task. All the three works I will present in this thesis required the creation of automated analysis tools, which will be described in details throughout this thesis.

1.1.3 Approach

Each of the three main works presented in this thesis has been carried out by using the following three-step approach:

1. First, I identified a component, within the mobile ecosystem, which is vulnerable to different types of attackers, and I systematically studied its attack surface.

2. Then, I performed large-scale, automated, studies to characterize the presence of the previously identified problems in real-world applications.

3. Finally, I proposed and implemented mitigations to the identified security issues.

The following thesis statement summarizes my motivations, approach, and results:

> *In the mobile ecosystem, different components trust each other. The consequences of these trust relationships are not well understood by both apps' developers and operating systems' designers.*
>
> *To solve this issue, first, we need to systematically study the attack surfaces of the different components trusting each other. Then, we need to perform automated large-scale studies to identify both benign apps, misplacing trust in untrusted components, and malicious apps, exploiting weaknesses in the ecosystem. Finally, we need to fix these weaknesses by better designing, implementing, and documenting these components.*

Table 1.1: Summary of the main contributions of the three works presented in this thesis, with respect to the aforementioned three-step approach.
SA indicates tools using Static Analysis, *DA* indicates tools using Dynamic Analysis

Chapter	Analyzed Components	Implemented Automated Analyses	Proposed/Implemented Solutions
2	– Users – User Interface – Apps	– *DA*: UI state exploration [2.3] – *SA*: Detection of apps performing GUI-confusions attacks [2.4]	– Market-Level detection [2.4] – UI modifications conveying the origin of the showed content [2.5]
3	– Apps – Operating System – Remote Backends	– *DA*: Identification of apps using "device-public" information for authentication [3.3]	– Secure SMS Channel [3.6.1] – Secure Device ID [3.6.2]
4	– Apps – Operating System – Remote Backends – Trusted Execution Environments	– *SA*: Fingerprint API usage classifier [4.6]	– Trusted UI for authentication [4.8.1] – UI modifications showing how TrustZone and the fingerprint reader sensor are used [4.8.2] – Improved key attestation mechanism [4.8.3]

1.2 Summary of the Main Contributions

In this thesis, I will present three main works. This section gives an overview of these works and their the main contributions. Table 1.1 summarizes their contributions with respect to the aforementioned three-step approach.

1.2.1 User/User-Interface Trust Relationship

Users are the *final consumers* of any operation carried on by a mobile device, and user interfaces mediate all the interactions between users and their devices. This observation motivated me to start my research on mobile security by focusing on the implementation of these user interfaces. In particular, I studied how it is possible, in the Android operating system, to lure users in interacting with malicious apps that *look like* legitimate ones. Specifically, an attacker who is able to install a malicious app on a user's device, can wait for the user to open any app requiring a login (e.g., the popular "Facebook" app) and cover the app's legitimate login interface with an attacker-controlled one, which leaks the inserted credentials to the attacker.

As part of this research project, I first systematically identified all the ways in which it is possible to perform this kind of attack. Due to the complexity of the Android user interface (UI) API and inadequacy in the Android documentation, even conceptually simple questions, such as "How can an attacker draw content on top of a legitimate app?", do not have an easy answer. Therefore, the systematization of the behavior of the Android UI required the development of automated dynamic analysis tools exploring the entire UI API's parameter space, looking for combinations of API calls and parameters allowing attackers to achieve their nefarious goals.

Afterward, I developed a static analysis system to detect malicious apps exploiting this issue. This system was able to automatically detect applications that interfered with the Android UI. As an example, one detected sample presented to the user a malicious interface asking for credit card information, mimicking the legitimate UI normally shown by the official Google Play Store.

Finally, I designed and implemented modifications to the Android UI to inform the users about the origin of the apps they are interacting with (e.g., the legitimate PayPal app is developed by "PayPal, Inc."). To evaluate this on-device defense mechanism, I performed a user-study involving hundreds of users. The results of this study showed that, while in a standalone system is trivial for an attacker to lure users to interact with a "spoofed" login interface, when the developed defense mechanism is in place, the majority of the users can detect when they are attacked.

1.2.2 Trusting the Operating System for Authentication

After addressing issues in the UI of mobile devices, I focused on analyzing how Android applications authenticate users while interacting with remote backends.

In particular, I was able to identify a class of authentication mechanisms which are *intrinsically* insecure. These authentication schemes are based on what I define as *device-public information*, which consists of properties and data that any application running on a device can obtain, such as the device's identifiers, files in publicly accessible locations, and the content of received text messages. While these schemes are convenient to users (since they require little or no interaction to perform authentication), they are vulnerable by design, because all the needed information to authenticate a user is available to any

app installed on a device. An attacker with a malicious app on a user's device could easily hijack the user's account, steal private information, send (and receive) messages on behalf of the user, or steal valuable virtual goods.

To understand how widespread this class of vulnerability is and to help developers and application market operators identify vulnerable apps, I developed a dynamic analysis system that aims to uncover apps affected by this problem. The developed system records the app's UI behavior during its first execution on a device, when authentication is likely to happen. Then, as a second step, the system wipes the app's private data and runs the app again, still recording its UI behavior. The key intuition is that if the UI behavior of an app changes after its re-installation, it means the app may rely on device-public information for authentication and is very likely to be vulnerable to this attack. This is due to the fact that re-installation deletes all the private data of an app. Therefore, if the user is still authenticated, it means that the authentication procedure must leverage device-public information. As a final step, the system attempts to confirm the vulnerability by exploiting it to login on behalf of the legitimate user from a different, attacker-controlled device, on which the device-public information present in the victim's device has been cloned.

I used this analysis system to vet 1,000 of the most popular applications from the Google Play Store, and found that 41 of them were vulnerable to this attack. Two of these vulnerable apps were WhatsApp and Viber, two of the most popular messaging apps, with hundreds of millions of installations. For both these apps, I discovered that it was sufficient to steal the content of a publicly accessible file (and spoof the value of some device's identifiers) to fully hijack a user account. I reported these findings to the respective security teams, which quickly acknowledged the problems and addressed the vulnerabilities.

1.2.3 Hardware-Assisted Authentication

Application developers strive to create authentication mechanisms that, unlike usernames and passwords, require minimal user interaction. Unfortunately, this can lead to unsafe authentication schemes, as the ones I mentioned before. Recently, new technologies have been developed to support authentication and authorization schemes that are both more usable and more secure than traditional username and password authentication.

In particular, most smartphones come with Trusted Execution Environments (TEEs) that can be used to generate and store cryptographic keys. Furthermore, TEEs can be programmed to directly communicate with a fingerprint reader sensor (which is widely available on modern smartphones). In this way, it is possible to build systems in which non-exportable, safely-stored cryptographic keys are only usable when the fingerprint reader detects a registered legitimate fingerprint, signaling the user's explicit consent to perform a specific security-sensitive operation, such as transferring money. Since a TEE is a hardware-enforced isolated execution environment, the keys it stores and the operations performed with those keys cannot be leaked or misused even if the smartphone's operating system is compromised. In addition, even if a device is stolen, an attacker cannot misuse the keys stored on it without the owner's fingerprint.

The combination of these factors allows, at least *in theory*, to implement authentication and authorization systems that provide strong security guarantees, even against powerful attackers who are able to compromise the operating system of a device. However, the reality is very different due to both the way in which most of the apps use the fingerprint API in Android and some weaknesses in its implementation.

To understand how developers use *in practice* this functionality, I performed the first systematic study on the usage of the fingerprint API in Android. This study was enabled by a static-analysis-based tool which I developed. The results are worrisome. For example, the tool identified that 53.69% of the 501 analyzed apps, including the widely deployed Google Play Store app, do not make use of the cryptographic keystore, unlocked by a legitimate fingerprint touch. As a consequence, an attacker having root privileges can completely bypass the fingerprint security mechanism simply by programmatically "simulating" the user's touch. Moreover, the current implementation of this API does not allow users to understand *what* they are authorizing by touching the fingerprint reader sensor. Therefore, an attacker can show an interface asking the user to touch the sensor to, for instance, "Transfer 100 dollars to *Friend A*," while, in reality, the user is authorizing the operation "Transfer 100 dollars to *Attacker*." To address this issue, I proposed modifications to the implementation of the Android fingerprint API and provided recommendations on how this API, and its documentation, should be improved to increase the number of developers using it correctly.

1.3 Permissions and Attributions

The content of Chapter 2 is the result of a collaboration with Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna, and part of a previously published paper [1].

The content of Chapter 3 is the result of a collaboration with Eric Gustafson, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna, and part of a previously published paper [2].

The content of Chapter 4 is the result of a collaboration with Yanick Fratantonio, Aravind Machiry, Christopher Kruegel, Giovanni Vigna, Simon Chung, and Wenke Lee, and part of a previously published paper [3].

Chapter 2

User/User-Interface Trust Relationship

Today, smartphone and tablet usage is on the rise, becoming the primary way of accessing digital media in the US [4]. Many users now trust their mobile devices to perform tasks, such as mobile banking or shopping, through mobile applications, typically called "apps." This wealth of confidential data has not gone unnoticed by cybercriminals: over the last few years, mobile malware has grown at an alarming rate [5].

Popular mobile operating systems run multiple apps concurrently. For example, a user can run both her mobile banking application and a new game she is checking out. Obviously, a game should not receive financial information. As a consequence, the ability to tell the two apps apart is crucial. At the same time, it is important for these apps to have user-friendly interfaces that make the most of the limited space and interaction possibilities.

Let us assume that a victim user is playing the game, which is malicious. When this user switches to another app, the game will remain active in the background (to support background processing and event notifications). However, it will also silently wait for the

user to login into her bank. When the malicious game detects that the user activates the banking app, it changes its own appearance to mimic the bank's user interface and instantly "steals the focus" to become the target with which the victim interacts. The user is oblivious to this switch of apps in the foreground, because she recognizes the graphical user interface (GUI) of the banking application. In fact, there have been no changes on the user's display throughout the attack at all, so it is impossible for her to detect it: she will then insert her personal banking credentials, which will then be collected by the author of the malicious app.

In the work presented in this chapter, we study this and a variety of other *GUI confusion* attacks. With this term, we denote attacks that exploit the user's inability to verify which app is, at any moment, drawing on the screen and receiving user inputs. GUI confusion attacks are similar to social engineering attacks such as phishing and click-jacking. As such, they are not fundamentally novel. However, we find that the combination of powerful app APIs and a limited user interface make these attacks much harder to detect on Android devices than their "cousins" launched on desktop machines, typically against web browsers.

The importance of GUI-related attacks on Android has been pointed out by several publications in the past, such as [6, 7] (with a focus on "tapjacking"), [8] (with a focus on phishing attacks deriving from control transfers), and [9] (with a focus on state disclosure through shared-memory counters). Our work generalizes these previously-discovered techniques by systematizing existing exploits. Furthermore, we introduce a number of novel attacks. As an extreme example of a novel attack, we found that a malicious app has the ability to create a complete virtual environment that acts as a full Android interface, with complete control of all user interactions and inputs. This makes it very hard for a victim user to escape the grip of such a malicious application. Even though at

the time of this writing the number of known samples performing GUI confusion attacks is limited, we believe that this is a real, currently unsolved, problem in the Android ecosystem.

This work also introduces two novel approaches to defend against GUI confusion attacks. The first approach leverages static code analysis to automatically find apps that could abuse Android APIs for GUI confusion attacks. We envision that this defense could be deployed at the market level, identifying suspicious apps before they hit the users. Interestingly, we detected that many benign apps are using potentially-dangerous APIs, thus ruling out simple API modifications as a defense mechanism.

Our static analysis approach is effective in identifying potentially-malicious apps. More precisely, our technique detects apps that interfere with the UI in response to some action taken by the user (or another app). The apps that we detect in this fashion fulfill two necessary preconditions of GUI confusion attacks: They monitor the user and other apps, and they interfere with the UI (e.g., by stealing the focus and occupying the top position on the screen). However, these two conditions are not sufficient for GUI confusion attacks. It is possible that legitimate apps monitor other apps and interfere with the UI. As an example, consider an "app-locker" program, which restricts access to certain parts of the phone (and other apps). When looking at the code, both types of programs (that is, malicious apps that launch GUI confusion attacks as well as app-lockers) look very similar and make use of the same Android APIs. The difference is in the intention of the apps, as well as the content they display to users. Malicious apps will attempt to mimic legitimate programs to entice the user to enter sensitive data. App-lockers, on the other hand, will display a screen that allows a user to enter a PIN or a password to unlock the phone. These semantic differences are a fundamental limitation for detection approaches that are purely code-based.

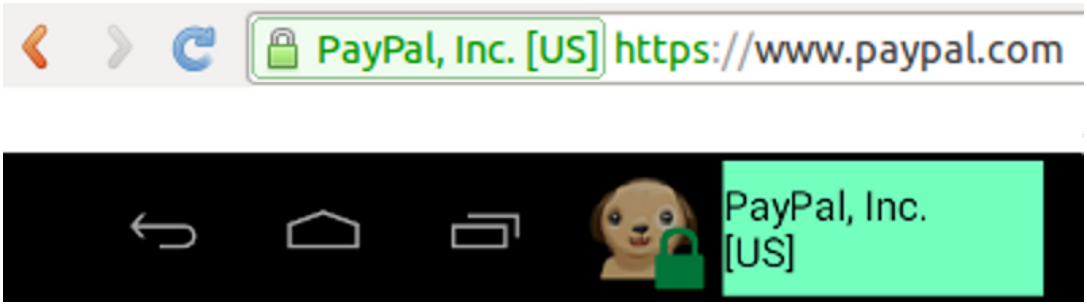


Figure 2.1: Comparison between how SSL Extended Validation information is shown in a modern Browser (Chrome 33) and what our implemented defense mechanism shows on the navigation bar of an Android device.

To address the limitations of code-based detection, we devised a second, on-device defense. This approach relies on modifications to the Android UI to display a trusted indicator that allows users to determine which app and developer they are interacting with, attempting to reuse security habits and training users might already have. To this end, we designed a solution (exemplified in Figure 2.1) that follows two well-accepted paradigms in web security:

- the Extended Validation SSL/TLS certification and visualization (the current-best-practice solution used by critical businesses to be safely identified by their users)
- the use of a "secure-image" to established a shared secret between the user interface and the user (similarly to what is currently used in different websites [10], [11] and recently proposed for the Android keyboard [12])

We evaluate the effectiveness of our solution with a user study involving 308 human subjects. We provided users with a system that implements several of our proposed defense modifications, and verified that the success ratio of the (normally invisible) deception attacks significantly decreases.

To summarize, the main contributions of this work are:

- We systematically study and categorize the different techniques an attacker can use to mount GUI deception attacks. We describe several new attack vectors that we found, and we introduce a tool to automatically explore reachable GUI states and identify the ones that can be used to mount an attack. This tool was able to automatically find two vulnerabilities in the Android framework that allow an app to gain full control of a device's UI.

- We study, using static analysis, how benign apps legitimately use API calls that render these attacks possible. Then, we develop a detection tool that can identify their malicious usage, so that suspicious apps can be detected at the market level.

- We propose an on-device defense that allows users to securely identify authors of the apps with which they interact. We compare our solution with the current state of the art, and we show that our solution has the highest coverage of possible attacks.

- In a user study with 308 subjects, we evaluate the effectiveness of these attack techniques, and show that our on-device defense helps users in identifying attacks.

For the source code of the proof-of-concept attacks we developed and the prototype of the proposed on-device defense, refer to our repository[1].

2.1 Background

To understand the attack and defense possibilities in the Android platform, it is necessary to introduce a few concepts and terms.

[1] https://github.com/ucsb-seclab/android_ui_deception

The Android platform is based on the Linux operating system and it has been designed mainly for touchscreen mobile devices. Unless otherwise noted, in this work we will mainly focus on Android version 4.4. When relevant, we will also explain new features and differences introduced by Android 5.0 (the latest available version at the time of writing).

In an Android device, apps are normally pre-installed or downloaded from the Google Play Store or from another manufacturer-managed market, although manual offline installation and unofficial markets can also be used. Typically, each app runs isolated from others except for well-defined communication channels.

Every app is contained in an *apk* file. The content of this file is signed to guarantee that the app has not been tampered with and that it is coming from the developer that owns the corresponding private key. There is no central authority, however, to ensure that the information contained in the developer's signing certificate is indeed accurate. Once installed on a device, an app is identified by its *package name*. It is not possible to install apps with the same package name at the same time on a single device.

Apps are composed of different developer-defined components. Specifically, four types of components exist in Android: *Activity*, *Service*, *Broadcast Receiver*, and *Content Provider*. An *Activity* defines a graphical user interface and its interactions with user's actions. Differently, a *Service* is a component running in background, performing long-running operations. A *Broadcast Receiver* is a component that responds to specific system-wide messages. Finally, a *Content Provider* is used to manage data shared with other components (either within the same app or with external ones).

To perform sensitive operations (e.g., tasks that can cost money or access private user data), apps need specific permissions. All the permissions requested by a *non-system* app must be approved by the user during the app's installation: a user can either grant all requested permissions or abort the installation. Some operations require permissions

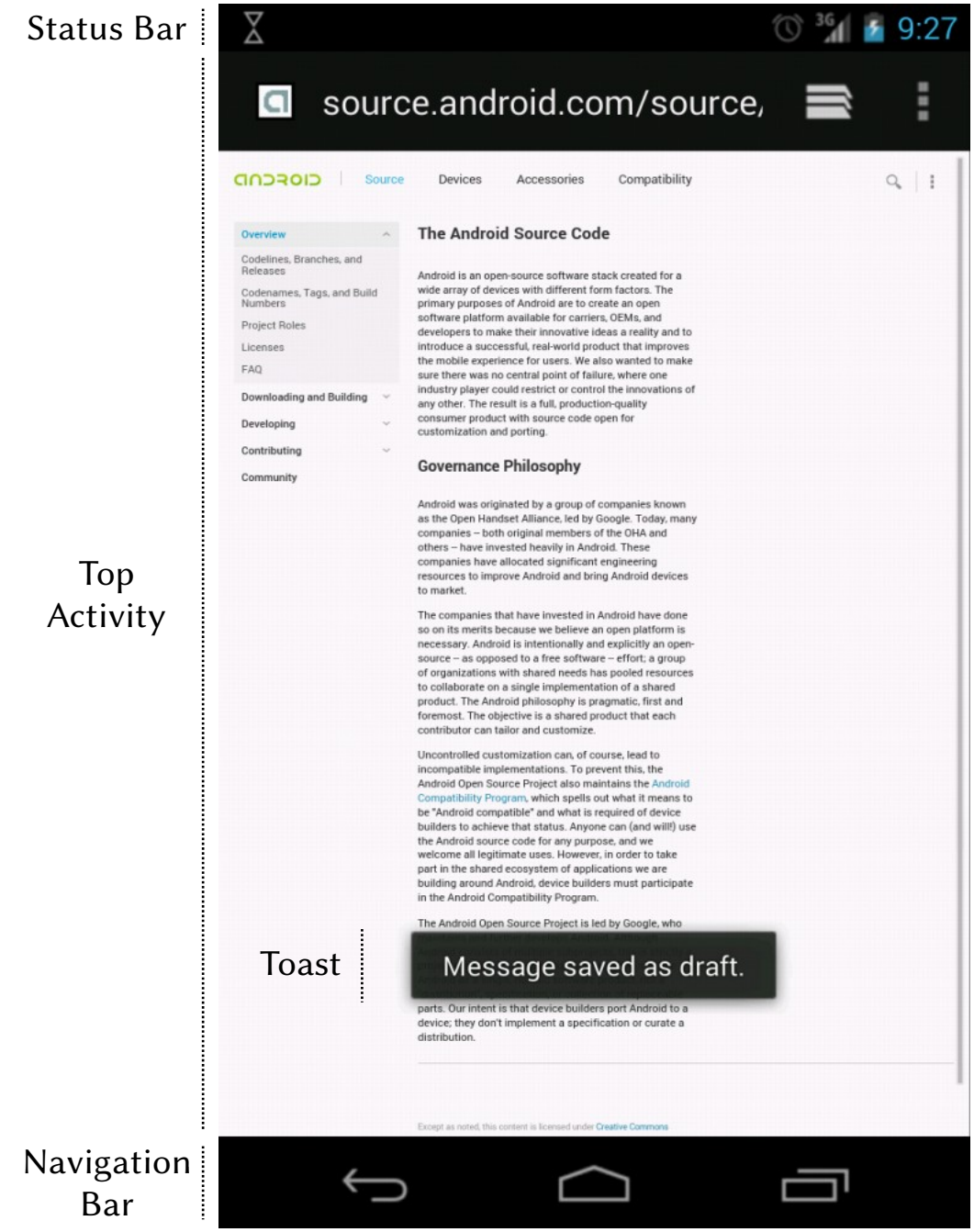


Figure 2.2: Typical Android user interface appearance. The status bar is at the top of the screen, while the navigation bar occupies the bottom. A browser app is open, and its main Activity is shown in the remaining space.

that are only granted to *system* apps (typically pre-installed or manufacturer-signed). Required permissions, together with other properties (such as the package name and the list of the app's components), are defined in a manifest file (`AndroidManifest.xml`), stored in the app's apk file.

2.1.1 Android graphical elements

Figure 2.2 shows the typical appearance of the Android user interface on a smartphone. The small *status bar*, at the top, shows information about the device's state, such as the current network connectivity status or the battery level. At the bottom, the *navigation bar* shows three big buttons that allow the user to "navigate" among all currently running apps as well as within the focused app.

Details may vary depending on the manufacturer (some devices merge the status and navigation bars, for instance, and legacy devices may use hardware buttons for the navigation bar). In this work we will use as reference the current guidelines[2], as they represent a typical modern implementation; in general, our considerations can be adapted to any Android device with minor modifications.

Apps draw graphical elements by instantiating system-provided components: Views, Windows, and Activities.

Views. A *View* is the basic UI building block in Android. Buttons, text fields, images, and OpenGL viewports are all examples of views. A collection of Views is itself a View, enabling hierarchical layouts.

[2] http://developer.android.com/design/handhelds/index.html, http://developer.android.com/design/patterns/compatibility.html

Activities. An *Activity* can be described as a controller in a Model-View-Controller pattern. An Activity is usually associated with a View (for the graphical layout) and defines actions that happen when the View elements are activated (e.g., a button gets clicked).

Activities are organized in a global stack that is managed by the *ActivityManager* system Service. The Activity on top of the stack is shown to the user. We will call this the *top Activity* and the app controlling it the *top app*.

Activities are added and removed from the Activity stack in many situations. Each app can reorder the ones it owns, but separate permissions are required for global monitoring or manipulation. Users can request an Activity switch using the navigation bar buttons:

- The *Back* button (bottom left in Figure 2.2) removes the top Activity from the top of the stack, so that the one below is displayed. This default behavior can be overridden by the top Activity.
- The *Home* button lets the user return to the base "home" screen, usually managed by a system app. A normal app can only replace the home screen if the user specifically allows this.
- The *Recent* button (bottom right in Figure 2.2) shows the list of top Activities of the running apps, so the user can switch among them. Activities have the option not to be listed. In Android 5.0, applications can also decide to show different thumbnails on the Recent menu (for instance, a browser can show a different thumbnail in the Recent menu for each opened tab).

Windows. A *Window* is a lower-level concept: a virtual surface where graphical content is drawn as defined by the contained Views. In Figure 2.2, the Status Bar, the Navigation Bar and the top Activity are all drawn in separate Windows. Normally, apps

do not explicitly create Windows; they just define and open Activities (which in turn define Views), and the content of the top Activity is drawn in the system-managed *top-activity Window*. Windows are normally managed automatically by the *WindowManager* system Service, although apps can also explicitly create Windows, as we will show later.

2.2 GUI confusion attacks

In this section, we discuss classes of GUI confusion attacks that allow for launching stealthy and effective phishing-style or click-jacking-style operations.

In our threat model, a malicious app is running on the victim's Android device, and it can only use APIs that are available to any benign non-system app. We will indicate when attacks require particular permissions. We also assume that the base Android operating system is not compromised, forming a Trusted Computing Base.

We have identified several Android functionalities (*Attack Vectors*, categorized in Table 2.1) that a malicious app can use to mount GUI confusion attacks. We have also identified *Enhancing Techniques*: abilities (such as monitoring other apps) that do not present a GUI security risk in themselves, but can assist in making attacks more convincing or stealthier.

2.2.1 Attack vectors

Draw on top

Attacks in this category aim to draw graphical elements over other apps. Typically, this is done by adding graphical elements in a Window placed over the top Activity. The Activity itself is not replaced, but malware can cover it either completely or partially and change the interpretation the user will give to certain elements.

Table 2.1: Attack vectors and enhancing techniques. We indicate with a dash attacks and techniques that, to the best of our knowledge, have not been already mentioned as useful in GUI confusion attacks.

Category	Attack vector	Mentioned in
Draw on top	UI-intercepting draw-over	[6, 8]
	Non-UI-intercepting draw-over	[6], [7], [8]
	Toast message	[6], [13]
App switch	*startActivity* API	[9]
	Screen pinning	—
	moveTaskTo APIs	—
	killBackgroundProcesses API	—
	Back / power button (passive)	—
	Sit and wait (passive)	—
Fullscreen	non-"immersive" fullscreen	—
	"immersive" fullscreen	—
	"inescapable" fullscreen	—
Enhancing techniques	*getRunningTask* API	[8]
	Reading the system log	[14]
	Accessing proc file system	[15, 9]
	App repackaging	[16], [17], [18]

Apps can explicitly open new Windows and draw content in them using the *addView* API exposed by the WindowManager Service. This API accepts several flags that determine how the new Window is shown (for a complete description, refer to the original documentation[3]). In particular, flags influence three different aspects of a Window:

- Whether it is intercepting user input or is letting it "pass through" to underlying Windows.

[3] http://developer.android.com/reference/android/view/WindowManager.LayoutParams.html

- Its *type*, which determines the Window's Z-order with respect to others.
- The region of the screen where it is drawn.

Non-system apps cannot open Windows of some types, while Windows with a higher Z-order than the top-activity Window require the *SYSTEM_ALERT_WINDOW* permission.

Windows used to display *toasts*, text messages shown for a limited amount of time, are an interesting exception. Intended to show small text messages even when unrelated apps control the main visualization, toast messages are usually created with specific APIs and placed by the system in Windows of type *TOAST*, drawn over the top-activity Window. No specific permission is necessary to show toast messages. Their malicious usage has been presented by previous research (refer to Table 2.1).

Two other types of attack are possible:

- *UI-intercepting draw-over:* A Window spawned using, for instance, the *PRIORITY_PHONE* flag can not only overlay the top-activity Window with arbitrary content, but also directly steal information by intercepting user input.
- *Non UI-intercepting draw-over:* By forwarding all user input to the underlying Windows, classical "click-jacking" attacks are possible. In these attacks, users are lured to perform an unwanted action while thinking they are interacting with a different element.

App switch

Attacks that belong to this category aim to steal focus from the top app. This is achieved when the malicious app seizes the top Activity: that is, the malicious app replaces the legitimate top Activity with one of its own. The malicious app that we

developed for our user study (Section 3.4) uses an attack in this category: it waits until the genuine Facebook app is the top app, and then triggers an app switch and changes its appearance to mimic the GUI of the original Facebook app.

Replacing the currently running app requires an *active* app switch. *Passive* app switches are also possible: in this case, the malicious application does not actively change the Activity stack, nor it shows new Windows, but it waits for specific user's input.

We have identified several attack vectors in this category:

***startActivity* API.** New Activities are opened using the *startActivity* API. Normally, the newly opened Activity does not appear on top of Activities of other apps. However, under particular conditions the spawned Activity will be drawn on top of all the existing ones (even if belonging to different apps) without requiring any permission. Three different aspects determine this behavior: the type of the Android component from which the *startActivity* API is called, the *launchMode* attribute of the opened Activity, and flags set when *startActivity* is called.

Given the thousands of different combinations influencing this behavior and the fact that the official documentation[4] does not state clearly when a newly Activity will be placed on top of other apps' Activities, we decided to develop a tool to systematically explore the conditions under which this happens.

Our tool determined that opening an Activity from a Service, a Broadcast Receiver, or a Content Provider will always place it on top of all the others, as long as the *NEW_TASK* flag is specified when the *startActivity* API is called. Alternatively, opening an Activity from another one will place the opened Activity on top of all the others if the *singleInstance* launch mode is specified. In addition, our tool found other, less common, situations in which an Activity is placed on top of all the others. For more details and a description of our tool, refer to Section 2.3.1.

[4] http://developer.android.com/guide/components/tasks-and-back-stack.html

***moveTaskTo* APIs.** Any app with the *REORDER_TASKS* permission can use the *moveTaskToFront* API to place Activities on top of the stack. We also found another API, *moveTaskToBack*, requiring the same permission, to remove another app from the top of the Activity stack.

Screen pinning. Android 5.0 introduces a new feature called "screen pinning" that locks the user interaction to a specific app. Specifically, while the screen is "pinned," there cannot be any switch to a different application (the Home button, the Recent button, and the status bar are hidden). Screen pinning can be either manually enabled by a user or programmatically requested by an app. In the latter case, user confirmation is necessary, unless the app is registered as a "device admin" (which, again, requires specific user confirmation).

***killBackgroundProcesses* API.** This API (requiring the *KILL_BACKGROUND_PROCESSES* permission) allows killing the processes spawned by another app. It can be used maliciously to interfere with how benign apps work: besides mimicking their interface, a malicious app could also prevent them from interacting with the user. Android does not allow killing the app controlling the top Activity, but other attack vectors can be used to first remove it from the top of the stack.

Back/Power Button. A malicious app can also make the user believe that an app switch has happened when, in fact, it has not. For example, an app can intercept the actions associated with the *back* button. When the user presses the back button, she expects one of two things: either the current app terminates, or the previous Activity on the stack is shown. A malicious app could change its GUI to mimic its target (such as a login page) in response to the user pressing the back button, while at the same time disabling the normal functionality of the back button. This might make the user

believe that an app switch has occurred, when, in fact, she is still interacting with the malicious app. A similar attack can be mounted when the user turns off the screen while the malicious app is the top app.

Sit and Wait. When a malicious app is in the background, it can change its GUI to that of a victim app, so that when the user switches between apps looking, for example, for the legitimate banking application, she could inadvertently switch to the malicious version instead. This type of attack is known in the browser world as *tabnabbing* [19].

Fullscreen

Android apps have the possibility to enter the so called *fullscreen mode*, through which they can draw on the device's entire screen area, including the area where the navigation bar is usually drawn. Without proper mitigations, this ability could be exploited by malicious apps, for example, to create a fake home screen including a fake status bar and a fake navigation bar. The malicious app would therefore give the user the impression she is interacting with the OS, whereas her inputs are still intercepted by the malicious app.

Android implements specific mitigations against this threat [20]: An app can draw an Activity on the entire screen, but in principle users always have an easy way to close it and switch to another app. Specifically, in Android versions up to 4.3, the navigation bar appears on top of a fullscreen Activity as soon as the user clicks on the device screen. Android 4.4 introduces a new "immersive" fullscreen mode in which an Activity remains in fullscreen mode during all interactions: in this case, the navigation bar is accessed by performing a specific "swipe" gesture.

Given the large number of possible combinations of flags that apps are allowed to use to determine the appearance of a Window in Android, these safety functionalities are intrinsically difficult to implement. In fact, the implementation of the Android APIs in

charge of the creation and display of Windows has thousands of lines of code, and bugs in this APIs are likely to enable GUI confusion attacks. Therefore, we used our API exploration tool to check if it is possible to create a Window that covers the entire device's screen area (including the navigation bar) without giving any possibility to the user to close it or to switch to another application. We call a Window with these properties an *"inescapable"* fullscreen Window.

Our tool works by spawning Windows with varying input values of GUI-related APIs and, after each invocation, determines whether an "inescapable" fullscreen mode is entered. By using it, several such combinations were found, thus leading to the discovery of vulnerabilities in different Android versions. Upon manual investigation, we found that Google committed a patch[5] to fix a bug present in Android 4.3; however, our tool pointed out that this fix does not cover all possible cases. In fact, we found a similar problem that affects Android versions 4.4 and 5.0. We notified Google's Security Team: a review is in progress at the time of this writing.

Section 2.3.2 presents more technical details about the tool we developed and its findings.

There is effectively no limit to what a malicious programmer can achieve using an "inescapable" fullscreen app. For instance, one can create a full "fake" environment that retains full control (and observation powers) while giving the illusion of interacting with a regular device (either by "proxying" app Windows or by relaying the entire I/O to and from a separate physical device).

2.2.2 Enhancing techniques

Additional techniques can be used in conjunction with the aforementioned attack vectors to mount more effective attacks.

[5] https://android.googlesource.com/platform/frameworks/base/+/b816bed

Techniques to detect how the user is currently interacting with the system

To use the described attack vectors more effectively, it is useful for an attacker to know how the user is currently interacting with the device.

For instance, suppose again that a malicious app wants to steal bank account credentials. The most effective way would be to wait until the user actually opens the specific login Activity in the original app and, immediately after, cover it with a fake one. To do so, it is necessary to know which Activity and which app the user is currently interacting with.

We have identified a number of ways to do so: some of them have been disabled in newer Android versions, but others can still be used in the latest available Android version.

Reading the system log. Android implements a system log where standard apps, as well as system Services, write logging and debugging information. This log is readable by any app having the relatively-common *READ_LOGS* permission (see Table 2.4 in the next section). By reading messages written by the ActivityManager Service, an app can learn about the last Activity that has been drawn on the screen.

Moreover, apps can write arbitrary messages into the system log and this is a common channel used by developers to receive debug information. We have observed that this message logging is very commonly left enabled even when apps are released to the public, and this may help attackers time their actions, better reproduce the status of an app, or even directly gather sensitive information if debug messages contain confidential data items.

Given the possible malicious usage of this functionality, an app can only read log messages created by itself in Android version 4.1 and above.

***getRunningTasks* API.** An app can get information about currently running apps by invoking the *getRunningTasks* API. In particular, it is possible to know which app is on top and the name of the top Activity. The relatively-common *GET_TASKS* permission is required to perform such queries.

The functionality of this API has been changed in Android 5.0, so that an app can only use it to get information about its own Activities. For this reason, in Android 5.0 this API cannot be used anymore to detect which application is currently on top.

Accessing the proc file system. It is possible to get similar information by reading data from the *proc* file system, as previous research [15, 9] studied in detail both in a generic Linux system and in the specific setup of an Android device.

For instance, an app can retrieve the list of running applications by listing the */proc* directory and reading the content of the file: */proc/<process_pid>/cmdline*. However, most of the apps have a process running in the background even when a user is not interacting with them, so this information cannot be used to detect the app showing the top Activity.

More interestingly, we have identified a technique to detect which is the app the user is currently interacting with. In particular, the content of the file */proc/<process_pid>/cgroups* changes (from "/apps/bg_non_interactive" to "/apps") when the app on top is run by the <process_pid>. This is due to the fact that Android (using Linux cgroups) uses the specific "/apps" scheduling category for the app showing the top activity. We have tested this technique in Android 5.0 and, to the best of our knowledge, we are the first one pointing out the usage of this technique for GUI-related attacks in Android.

Finally, as studied in [9], by reading the content of */proc/<process_pid>/statm*, an application can infer the graphical state of another app, and precisely identify the specific Activity with which a user is interacting.

Techniques to create graphical elements mimicking already existing ones

To effectively replace an Activity of a "victim app," a convincing copy is necessary. Of course, an attacker could develop a malicious app from scratch with the same graphical elements as the original one. However, it is also possible to take the original app, change its package name, and just add the attack and information-gathering code.

The procedure of modifying an existing app (called *repackaging*) is well-known in the Android ecosystem. In the context of this work, repackaging is a useful technique to expedite development of interfaces that mimic those of other apps. Note, however, that the attacks described in this section are entirely possible without repackaging. Detecting and defending from repackaging is outside the scope of this work.

2.2.3 Attack app examples

In practice, malicious apps can combine multiple attack vectors and enhancing techniques to mount stealthy attacks. For instance, the attack app we implemented for our user study portraits itself as a utility app. When launched, it starts to monitor other running apps, waiting until the user switches to (or launches) the Facebook app. When that happens, it uses the *startActivity* API to spawn a malicious app on top of the genuine Facebook app. The malicious app is a repackaged version of the actual Facebook app, with the additional functionality that it leaks any entered user credentials to a remote location. To be stealthier, it informs Android that it should not be listed in the Recent Apps view.

We also developed a proof-of-concept malicious app that covers and mimics the home screen of a device, and demonstration videos. The displayed attack uses the "immersive" fullscreen functionality, but it can be easily adapted to use the "inescapable" fullscreen mode described in Section 2.2.1.

2.3 State exploration of the Android GUI API

We have developed a tool to study how the main Android GUI APIs can be used to mount a GUI confusion attack. The tool automatically performs a full state exploration of the parameters of the *startActivity* API, which can be used to open Activities on top of others (including Activities of different apps). Also, our tool systematically explores all Window-drawing possibilities, to check if it is possible to create Windows that:

1. entirely cover the device's screen;

2. leave the user no way to close them or access the navigation bar.

In the following two sections, we will explain our tool in detail, and we will show what it has automatically found.

2.3.1 Study of the *startActivity* API

First, using the documentation and the source code as references, we determined that three different aspects influence how a newly-started Activity is placed on the Activities' stack:

- The type of Android component calling *startActivity*.
- The *launchMode* attribute of the opened Activity.
- Flags passed to *startActivity*.

Table 2.2 lists the possible Android component types, all the relevant flags and *launchMode* values an app can use.

Our tool works by first opening a "victim" app that controls the top Activity. A different "attacker" app then opens a new Activity calling the *startActivity* API with every possible combination of the listed launch modes and flags. This API is called

Table 2.2: Component types, flags, and *launchMode* values tested by our tool

Component type	Activity, Service,Content Provider, Broadcast Receiver
launchMode attribute	standard, singleTop, singleTask, singleInstance
startActivity flags	MULTIPLE_TASK, NEW_TASK, CLEAR_TASK, CLEAR_TOP, PREVIOUS_IS_TOP, REORDER_TO_FRONT, SINGLE_TOP, TASK_ON_HOME

in four different code locations, corresponding to the four different types of Android components. Our tool then checks if the newly-opened Activity has been placed on top of the "victim" app, by taking a screenshot and analyzing the captured image.

Our tool found, in Android version 4.4, the following three conditions under which an Activity is drawn on top of every other:

1. The Activity is opened by calling the *startActivity* API from a Service, a Broadcast Receiver, or a Content Provider and the *NEW_TASK* flag is used.

2. The Activity is opened by calling the *startActivity* API from another Activity and it has the *singleInstance* launch mode.

3. The Activity is opened by calling the *startActivity* API from another Activity and one of the following combinations of launch modes and flags is used:

   - *NEW_TASK* and *CLEAR_TASK* flags.
   - *NEW_TASK* and *MULTIPLE_TASK* flags, and launch mode different from *singleTask*.
   - *CLEAR_TASK* flag and *singleTask* launch mode.

Table 2.3: Window types and flags. Flags in *italics* are only available starting from Android version 4.4, whereas TYPEs in **bold** require the SYSTEM_ALERT_WINDOW permission.

TYPEs	TOAST, **SYSTEM_ERROR**, **PHONE**, **PRIORITY_PHONE**, **SYSTEM_ALERT**, **SYSTEM_OVERLAY**
Layout flags	IN_SCREEN, NO_LIMITS,
System-UI Visibility flags	HIDE_NAVIGATION, FULLSCREEN, LAYOUT_HIDE_NAVIGATION, LAYOUT_FULLSCREEN, *IMMERSIVE*, *IMMERSIVE_STICKY*

We are only aware of one previous paper [9] that (manually) studies the behavior of this API for different parameters and under different conditions. Interestingly, the authors do not find all the conditions that we discovered. This underlines how the complexity of the Android API and omissions in the official documentation are prone to creating unexpected behaviors that are triggered using undocumented combinations of flags and APIs. Such behaviors are hard to completely cover through manual investigation. Hence, our API exploration tool can effectively help Android developers to detect these situations. As one example, we will now discuss how our tool revealed the existence of an "inescapable" fullscreen possibility.

2.3.2 Study of "inescapable" fullscreen Windows

We first checked the documentation and source code to determine the three different ways in which an app can influence the appearance of a Window that are relevant to our analysis:

- Modifying the Window's *TYPE*.
- Specifying certain flags that determine the Window's layout.

- Calling the *setSystemUiVisibility* API with specific flags to influence the appearance and the behavior of the navigation bar and the status bar.

Table 2.3 lists all the relevant flags and Window types an app can use.

Our tool automatically spawns Windows with every possible combination of the listed types and flags. After spawning each Window, it injects user input that should close a fullscreen Window, according to the Android documentation (e.g., a "slide" touch from the top of the screen). It then checks if, after the injection of these events, the Window is still covering the entire screen, by taking a screenshot and analyzing the captured image.

Using our tool we were able to find ways to create an "inescapable" fullscreen Window in Android 4.3, 4.4 and 5.0, which we will now briefly describe.

In particular, a Window of type SYSTEM_ERROR created with the flag NO_LIMITS, can cover the device's entire screen in Android 4.3. To specifically address this problem, a patch has been committed in the Android code before the release of the version 4.4. This patch limits the position and the size of a Window (so that it cannot cover the navigation bar) if it has this specific combination of type and flag.

However, this patch does not cover all the cases. In fact, the "immersive" fullscreen mode introduced in Android 4.4 opens additional ways to create "inescapable" fullscreen Windows, such as using the SYSTEM_ERROR type and then calling the *setSystemUiVisibility* API to set the LAYOUT_HIDE_NAVIGATION, HIDE_NAVIGATION, LAYOUT_FULLSCREEN, and IMMERSIVE_STICKY flags. We verified that the same parameters create an "inescapable" fullscreen Window in Android 5.0 as well.

It is important to notice that all the ways we discovered to create "inescapable" fullscreen Windows require using the SYSTEM_ERROR type. To fully address this problem, we propose removing this type or restricting its usage only to system components.

2.4 Detection via static analysis

We developed a static analysis tool to explore how (and whether) real-world apps make use of the attack vectors and enhancing techniques that we previously explained in Section 2.2. Our goals with this tool are two-fold:

1. Study if and how the techniques described in Section 2.2 are used by benign apps and/or by malicious apps, to guide our defense design.

2. Automatically detect potentially-malicious usage of such techniques.

2.4.1 Tool description

Our tool takes as input an app's *apk* file and outputs a summary of the potentially-malicious techniques that it uses. In addition, it flags an app as *potentially-malicious* if it detects that the analyzed app has the ability to perform GUI confusion attacks.

Specifically, it first checks which permissions the app requires in its manifest. It then extracts and parses the app's bytecode, and it identifies all the invocations to the APIs related to the previously-described attack techniques. Then, the tool applies backward program slicing techniques to check the possible values of the arguments for the identified API calls. The results of the static analyzer are then used to determine whether a particular technique (or a combination of them) is used by a given application. Finally, by analyzing the app's control flow, it decides whether to flag it as (potentially) malicious.

In this section, we will discuss the static analyzer, the attack techniques that we can automatically detect, and the results we obtained by running the tool on a test corpus of over two thousand apps. We would like to note that the implementation of the basic

static analysis tool (namely, the backward program slicer) is not a contribution of this work: We reused the one that Egele et al. developed for Cryptolint [21], whose source code was kindly shared with us.

Program slicer

The slicer first decompiles the Dalvik bytecode of a given app by using Androguard [22]. It then constructs an over-approximation of the application's call graph representing all possible method invocations among different methods in the analyzed app. Then, a backward slicing algorithm (based on [23]) is used to compute slices of the analyzed app. Given an instruction I and a register R, the slicer returns a set of instructions that can possibly influence the value of R. The slice is computed by recursively following the def-use chain of instructions defining R, starting from instruction I. If the beginning of a method is reached, the previously-computed call graph is used to identify all possible calling locations of that method. Similarly, when a relevant register is the return value of another method call, the backward slicer recursively continues its analysis from the return instruction of the invoked method, according to the call graph.

As most of the static analysis tools focusing on Android, the slicer may return incomplete results if reflection, class loading, or native code are used. Dealing with such techniques is outside the scope of this project.

Detecting potential attack techniques

In the following, we describe how our tool identifies the different attack vectors and enhancing techniques.

Draw on top. We detect if the *addView* API, used to create custom Windows, is invoked with values of the *TYPE* parameter that give to the newly-created Window a Z-order higher than that of the top-activity Window.

In addition, to detect potentially-malicious usage of a toast message, we first look for all the code locations where a toast message is shown, and then we use the slicer to check if the *setView* API is used to customize the appearance of the message. Finally, we analyze the control flow graph of the method where the message is shown to detect if it is called in a loop. In fact, to create a toast message that appears as a persistent Window, it is necessary to call the *show* API repeatedly.

App Switch. Our tool checks if:

- The *startActivity* API is used to open an Activity that will be shown on top of others. As we already mentioned, three aspects influence this behavior: the type of the Android component from which the *startActivity* API is called, the *launchMode* attribute of the opened Activity, and flags set when *startActivity* is called. We determine the first aspect by analyzing the call graph of the app, the *launchMode* is read from the app's manifest file, whereas the used flags are detected by analyzing the slice of instructions influencing the call to the *startActivity* API.
- The *moveTaskToFront* API is used.
- The *killBackgroundProcesses* API is used.

We do not use as a feature the fact that an app is intercepting the back or power buttons, as these behaviors are too frequent in benign apps and, being *passive* methods, they have limited effectiveness compared to other techniques.

Fullscreen. Our tool checks if the *setUiVisibility* API is called with flags that cause it to hide the navigation bar.

Getting information about the device state. Our tool checks if:

- The *getRunningTasks* API is used.

- The app reads from the system log. Specifically, since the native utility *logcat* is normally used for this purpose, we check if the *Runtime.exec* API is called specifying the string "logcat" as parameter.

- The app accesses files in the */proc* file system. We detect this by looking for string constants starting with "/proc" within the app.

We did not use as a feature the fact that an app is a repackaged version of another, as its usage, even if popular among malware, is not necessary for GUI confusion attacks. If desired, our system can be completed with detection methods as those presented in [16, 17].

During our study, we found that some apps do not ask (on installation) for the permissions that would be necessary to call certain APIs for which we found calls in their code. For instance, we found some applications that contain calls to the *getRunningTask* API, without having the *GET_TASKS* permission. The reason behind this interesting behavior is that this API is called by library code that was included (but never used) in the app.

In the threat model we consider for this work, we assume that the Android security mechanisms are not violated. So, calling an API that requires a specific permission will fail if the app does not have it. For this reason, we do not consider an app as using one of the analyzed techniques if it lacks the necessary permissions.

Since the version 5.0 of Android has been released too close to the time of this research was carried out, we expect only a very limited (and not statistically significant) number of applications using techniques introduced in this version. For this reason, we decided not to implement the detection of the techniques only available in Android 5.0.

App classification. We classify an app as suspicious if the following three conditions hold:

1. The app uses a technique to get information about the device state.

2. The app uses an attack vector (any of the techniques in the Draw on top, App Switch, Fullscreen categories)

3. There is a path in the call graph of the app where Condition 1 (check on the running apps) happens, and then Condition 2 (the attack vector) happens.

Intuitively, the idea behind our classification approach is that, to perform an effective attack, a malicious app needs to decide when to attack (Condition 1) and then how to attack (Condition 2). Also, the check for when an attack should happen is expected to influence the actual launch of this attack (hence, there is a control-flow dependency of the attack on the preceding check, captured by Condition 3).

It is important to note that our tool (and the classification rules) are designed to identify the necessary conditions to perform a GUI confusion attack. That is, we expect our tool to detect any app that launches a GUI confusion attack. However, our classification rules are not sufficient for GUI confusion attacks. In particular, it is possible that our tool finds a *legitimate* app that fulfills our static analysis criteria for GUI confusion attacks. Consider, for example, applications of the "app-locker" category. These apps exhibit a behavior that is very similar to the attacks described in Section 2.2. They can be configured to "securely lock" (that is, disable) certain other apps unless a user-defined password is inserted. To this end, they continuously monitor running applications to check if one of the "locked" apps is opened and, when this happens, they cover it with a screen asking for an unlock password. At the code level, there is no difference between such apps and malicious programs. The difference is in the intent of the program, and the content shown to users when the app takes control of the screen.

We envision that our tool can be used during the market-level vetting process to spot apps that need manual analysis since they could be performing GUI confusion attacks. App-lockers would definitely need this analysis to check whether they are behaving according to their specification. In the following evaluation, we do not count app-lockers and similar programs as false positives. Instead, our system has properly detected an app that implements functionality that is similar to (and necessary for) GUI confusion attacks. The final decision about the presence of a GUI confusion attack has to be made by a human analyst. The reason is that static code analysis is fundamentally unable to match the general behavior of an app (and the content that it displays) to user expectations. Nonetheless, we consider our static analysis approach to be a powerful addition to the arsenal of tools that an app store can leverage. This is particularly true under the assumption that the number of legitimate apps that trigger our static detection is small. Fortunately, as shown in the next section, this assumption seems to hold, considering that only 0.4% of randomly chosen apps trigger our detection. Thus, our tool can help analysts to focus their efforts as part of the app store's manual vetting process.

One possibility to address the fundamental problem of static code analysis is to look at the app description in the market[6]. However, this approach is prone to miss malicious apps, as cybercriminals can deceive the detection system with a carefully-crafted description (i.e., disguising their password-stealer app as an app-locker).

A second possibility to address this fundamental problem is to devise a defense mechanism that empowers users to make proper decisions. One proposal for such a defense solution is based on the idea of a trusted indicator on the device that reliably and continuously informs a user about the application with which she is interacting. We will discuss the details of this solution in Section 2.5.

[6]A similar concept has been explored in Whyper [24], a tool to examine whether app descriptions indicate the reason why specific permissions are required.

Table 2.4: Number of apps requesting permissions used by GUI confusion attacks and number of apps using each detected technique in the analyzed data sets

permission name	***benign1* set**		***benign2* set**		***malicious* set**		***app-locker* set**	
GET_TASKS	32	6.4%	80	16.0%	217	17.2%	19	95.0%
READ_LOGS	9	1.8%	35	7.0%	240	19.1%	13	65.0%
KILL_BACKGROUND_PROCESSES	3	0.6%	13	2.6%	13	1.0%	5	25.0%
SYSTEM_ALERT_WINDOW	1	0.2%	34	6.8%	3	0.2%	10	50.0%
REORDER_TASKS	0	0.0%	4	0.8%	2	0.2%	2	10.0%

technique	***benign1* set**		***benign2* set**		***malicious* set**		***app-locker* set**	
startActivity API	53	10.6%	135	27.0%	751	59.6%	20	100.0%
killBackgroundProcesses API	1	0.2%	8	1.6%	6	0.5%	4	20.0%
fullscreen	0	0.0%	22	4.4%	0	0.0%	1	5.0%
moveToFront API	0	0.0%	0	0.0%	1	0.1%	1	5.0%
draw over using *addView* API	0	0.0%	9	1.8%	0	0.0%	3	15.0%
custom toast message	0	0.0%	1	0.2%	0	0.0%	1	5.0%
getRunningTasks API	23	4.6%	68	13.6%	147	11.7%	19	95.0%
reading from the system log	8	1.6%	18	3.6%	28	2.2%	8	40.0%
reading from *proc* file system	3	0.6%	26	5.2%	43	3.4%	4	20.0%

Table 2.5: Detection of potential GUI confusion attacks.

Dataset	Total	Detected	Correctly Detected	Notes
benign1 set	500	2	2	The detected apps are both app-lockers.
benign2 set	500	26	23	10 chat/voip app (jumping on top on an incoming phone call/message), 4 games (with disruptive ads), 4 enhancers (background apps monitoring and killing, persistent on-screen icon over any app), 2 anti-virus programs (jumping on top when a malicious app is detected), 2 app-lockers, and 1 keyboard (jumping on top to offer a paid upgrade).
app-locker set	20	18	18	Of the two we are not detecting, one is currently inoperable, and the other has a data dependency between checking the running apps and launching the attack (we only check for dependency in the control flow).
malicious set	1,260	25	21	21 of the detected apps belong to the *DroidKungFu* malware family, which aggressively displays an Activity on top of any other.

2.4.2 Results

We ran our tool on the following four sets of apps:

1. A set of 500 apps downloaded randomly from the Google Play Store (later called *benign1*).

2. A set of 500 apps downloaded from the "top free" category on the Google Play Store (later called *benign2*).

3. A set of 20 apps described as app-lockers in the Google Play Store (later called *app-locker*).

4. A set of 1,260 apps from the Android Malware Genome project[25] (later called *malicious*).

The top part of Table 2.4 shows the usage of five key permissions that apps would need to request to carry out various GUI confusion attacks, for each of the four different data sets we used to evaluate our tool. From this data, it is clear that three out of five permissions are frequently used by benign applications. As a result, solely checking for permissions that are needed to launch attacks cannot serve as the basis for detection, since they are too common.

The bottom part of Table 2.4 details how frequently apps call APIs associated with the different techniques. Again, just looking at API calls is not enough for detection. Consider a simplistic (*grep-style*) approach that detects an app as suspicious when it uses, at least once, an API to get information about the state of the device and one to perform an attack vector. This would result in an unacceptable number of incorrect detections. Specifically, this approach would result in classifying as suspicious 33 apps in the *benign1* (6.6%) set and 95 in the *benign2* set (19.0%).

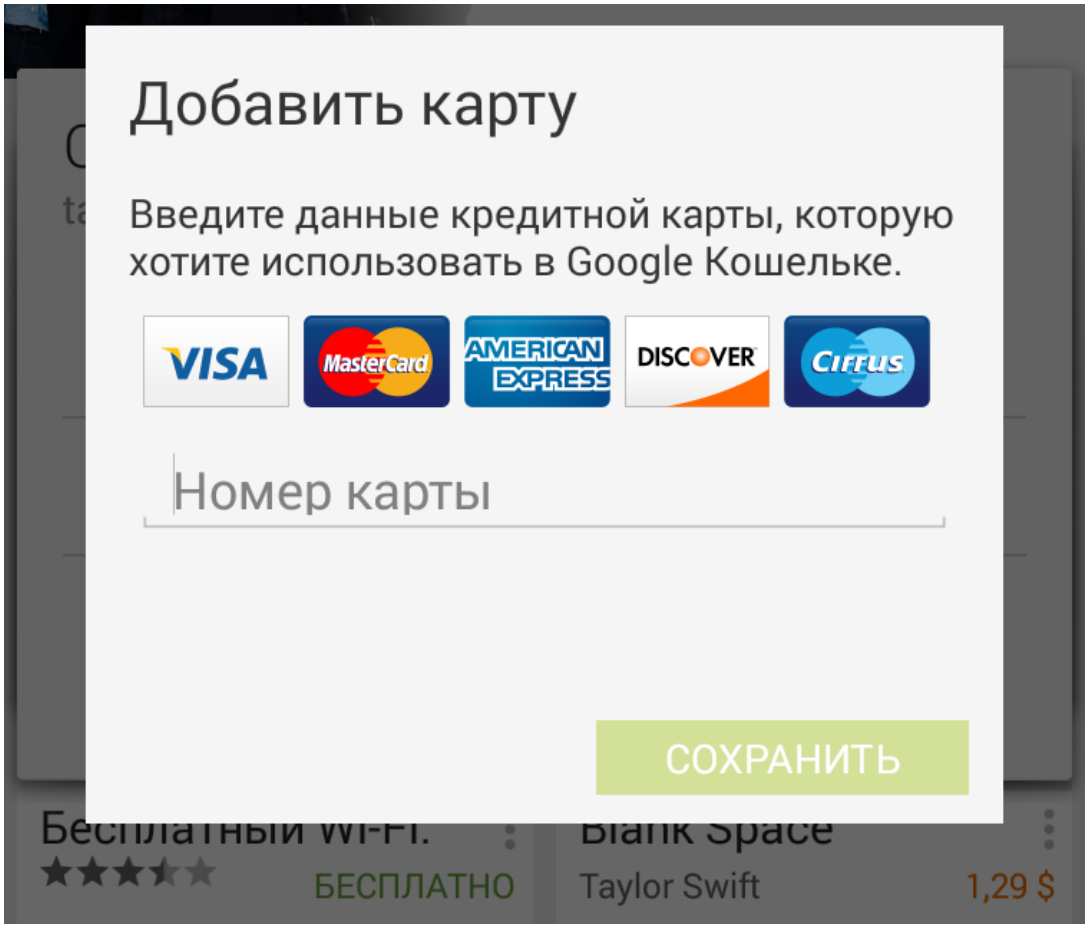


Figure 2.3: A screenshot acquired while the sample of the *svpeng* malware family, detected by our tool, is attacking the user. The Activity shown in the picture (asking, in Russian, to insert credit card information) is spawned by the malware while the user is on the official Google Play Store. Data entered in this Activity is then sent to a malicious server.

On the *benign1* set, our tool flagged two apps as suspicious. Manual investigation revealed that these applications monitor the user's Activity and, under specific conditions, block normal user interaction with the device. Even though these samples do not perform a GUI confusion attack (since they do not mimic the appearance of another application), they are both app-lockers. Hence, we expect our tool to report them.

On the *benign2* set, the tool detected 26 applications. When reviewing these apps, we found that two of them are app-lockers, ten of them are chat or VOIP apps, which display custom notifications using a separate mechanism than the status bar (such as stealing focus on an incoming phone call), four are games with disruptive ads, and four are "performance enhancers" (which monitor and kill the background running apps and keep a persistent icon on the screen). We also detected two anti-virus programs (which jump on top when a malicious app is detected) and one (annoying) keyboard app that jumps on top to offer a paid upgrade. We also had three false positives; two apps that could be used to take pictures, and one browser. These three apps satisfy the three conditions used to flag an app as potentially-malicious, but they do not interfere with the device's GUI.

The difference between results on sets *benign2* and *benign1* is due to the fact that popular apps are significantly bigger and more complex than the randomly-selected ones. In general, they do more and call a larger variety of APIs. Nonetheless, the total number of apps that would need to be manually analyzed is small, especially considering the set of random apps. Hence, an app store could use our system to perform a pre-filtering to check for apps that can potentially launch GUI confusion attacks, and then use manual analysis to confirm (or refute) this hypothesis.

To evaluate the detection capabilities (and false negative rate) of our tool, we randomly downloaded from the Google Play Store a set of 20 apps (called *app-locker*), described as app-lockers on the store. Since, as previously explained, this category of applications exhibits a behavior that is very similar to the attacks described in Section 2.2, we expected our tool to detect them all. Our tool detected 18 out of 20 samples. Manual investigation revealed that of the two undetected samples, one is currently inoperable and the other has a data dependency between checking the running apps and launching the attack (we only check for dependency in the control flow).

Finally, we tested our tool on the *malicious* set of 1,260 apps from the Android Malware Genome project [25]. Overall, most current Android malware is trying to surreptitiously steal and exfiltrate data, trying hard to remain unnoticed. Hence, we would not expect many samples to trigger our detection. In this set, we detected 25 apps as suspicious. Upon manual review, we found that 21 of the detected samples belong to the malware family *DroidKungFu*. These samples aggressively display an Activity on top of any other, asking to the user to either grant them "superuser" privileges or enable the "USB debugging" functionality (so that the root exploit they use can work). Due to code obfuscation, we could not confirm whether the other four samples were correct detections or not. To be on the safe side, we count them as incorrect detections.

We also ran our tool on a sample of the *svpeng* [26] malware family. To the best of our knowledge, this is the only Android malware family that currently performs GUI confusion attacks. Specifically, this sample detects when the official Google Play Store is opened. At this point, as shown in Figure 2.3, the malicious sample spawns an Activity, mimicking the original "Enter card details" Activity. As expected, our tool was able to detect this malicious sample. Furthermore, we tested our tool on an Android ransomware sample known to interfere with the GUI (*Android.Fakedefender*). As expected, our tool correctly flagged the app as suspicious, since it uses an enhancing technique (detecting if the user is trying to uninstall it) and an attack vector (going on top of the uninstall Activity to prevent users from using it).

Finally, we used our tool to check for the "inescapable" fullscreen technique. Our tool did not find evidence of its usage in any of the analyzed sets. This suggests that removing the possibility of using this very specific functionality (as we will propose in the next section) will not break compatibility with existing applications.

2.5 UI Defense mechanism

As mentioned, we complete our defense approach with a system designed to inform users and leave the final decision to them, exploiting the fact that the Android system is not being fooled by GUI attacks: Recall from Section 2.1.1 that all user-visible elements are created and managed via explicit app-OS interactions.

What compromises user security (and we consider the root cause of our attacks) is that there is simply no way for the user to know with which application she is actually interacting. To rectify this situation, we propose a set of simple modifications to the Android system to establish a trusted path to inform the user without compromising UI functionality.

In particular, our proposed modifications need to address three different challenges:

1. Understanding with which app the user is actually interacting.
2. Understanding who the real author of that app is.
3. Showing this information to the user in an unobtrusive but reliable and non-manipulable way.

Table 2.6: Examples of deception methods and whether defense systems protect against them.

	Fernandes et al. [12]	**Chen et al. [9]**	**Our on-device defense**
Keyboard input to the wrong app	✓	✗	✓
Custom input method to the wrong app (i.e., Google Wallet's PIN entry), on-screen info from the wrong app	Off by default, requires user interaction: The protection is activated only if the user presses a specific key combination.	✗	✓
Covert app switch	Keyboard only	✓ (animation)	✓
Faked app switch (through the back or power button)	Keyboard only	✗	✓
"Sit and Wait" (passive appearance change)	Keyboard only	✗	✓
Similar-looking app icon and name, installed through the market	✗ (the security indicator displays the similar-looking app icon and name. No verification of the author of the app happens.)	✗	✓
Side-loaded app, with the same app icon and name (possibly, through repackaging)	✗ (the security indicator displays the original app icon and name. No verification of the author of the app happens.)	✗	✓
Confusing GUI elements added by other apps (intercepting or non-intercepting draw-over, toast messages)	Off by default, requires user interaction	✗	✓ (yellow lock)
Presenting deceptive elements in non-immersive fullscreen mode	Off by default, requires user interaction	✗	✓
Presenting deceptive elements in immersive fullscreen mode	Off by default, requires user interaction	✗	✓ ("secret image")

Three independent components address these challenges. The combination of the states of components one and two determines the information presented to the user by component three.

Overall, two principles guided our choices:

- Offering security guarantees comparable with how a modern browser presents a critical (i.e., banking) website, identifying it during the entire interaction and presenting standard and recognizable visual elements.

- Allowing benign apps to continue functioning as if our defense were not in place, and not burdening the user with extra operations such as continuously using extra button combinations or requiring specific hardware modifications.

In particular, we wish to present security-conscious users with a familiar environment consistent with their training, using the same principles that brought different browser manufacturers to present similar elements for HTTPS-protected sites without hiding them behind browser-specific interactions.

An overview of the possible cases, how our system behaves for each of them, and the analogy with the web browser world that inspired our choices is presented in Table 2.7, while a more detailed description of each of our three components will be presented in the following sections.

Our implementation will be briefly described in Section 2.5.4, whereas Table 2.6 exemplifies deception methods and recaps how users are defended by our system and those described in [12] and [9], which target attacks similar to the ones we described (Section 5.1 provides more details).

2.5.1 Which app is the user interacting with?

Normally, the top Activity (and, therefore, the top app) is the target of user interaction, with two important exceptions:

1. Utility components such as the navigation bar and the status bar (Section 2.1.1) are drawn separately by the system in specific Windows.

2. An app, even if not currently on top of the Activity stack, can direct a separate Window to be drawn over the top-activity Window.

Interactions with utility components are very common and directly mediated by the system. Thus, we can safely assume that no cross-app interference can be created (the "Back" button in the navigation bar, for instance, is exclusively controlled by the top Activity) and we don't need to consider them (Point 1) in our defense.

However, as exemplified in Section 2.2, Windows shown by different apps (Point 2) can interfere with the ability of a user to interact correctly with the top app.

While we could prohibit their creation (and thus remove row 3 of Table 2.7), the ability to create "always-visible" Windows is used by common benign apps: for instance, the "Facebook Messenger" app provides the ability to chat while using other apps and it is currently the most popular free app on the Google Play Store. Therefore, we have decided to simply alert users of the fact that a second app is drawing on top of the current top app, and leave them free to decide whether they want this cross-app interaction or not.

Table 2.7: Possible screen states and how they are visualized.

if	*then*			
	Resulting UI state	**Visualization**	**Equivalent in browsers**	**Visualization in browsers**
no domain specified in the manifest	Apps not associated with any organization	Regular black navigation bar	Regular HTTP pages	no lock icon
Domain specified in the manifest, successful verification, *no* visible Windows from other apps	Sure interaction with a verified app	Green lock and company name	HTTPS verified page	Green lock, domain name, and (optionally) company name
Domain specified in the manifest, successful verification, visible Windows from other apps	Likely interaction with a verified app, but external elements are present	Yellow half-open lock	Mixed HTTP and HTTPS content	Varies with browsers, a yellow warning sign is common
Domain specified in the manifest, unknown validity,	Incomplete verification (networking issues)	Red warning page, user allowed to proceed	Self-signed or missing CA certificate	Usually, red warning page, user allowed to proceed
(other cases)	Failed verification	Red error page	Failed verification	Red error page

The official Android system also provides a limited defense mechanism:

1. As mentioned, a specific permission is necessary to create always-visible custom Windows. If it is granted during installation, no other checks are performed. It is impossible for the top app to prevent extraneous content from being drawn over its own Activities. *Toasts* are handled separately and do not require extra permissions.

2. The top app can use the *filterTouchesWhenObscured* API on its Views (or override the *onFilterTouchEventForSecurity* method) to prevent user input when content from other apps is present at the click location.

Given the attack possibilities, however, these defenses are not exhaustive for our purposes if not supplemented by the extra visualization we propose, as they still allow any extraneous content to be present over the top Activity. Moreover, the protection API can create surprising incompatibilities with benign apps (such as "screen darkeners") that use semi-transparent Windows, and does not prevent other apps' Windows from intercepting interactions (that is, it can protect only from Windows that "pass through" input).

The Android API could also be extended to provide more information and leave developers responsible to defend their own apps, but providing a defense mechanism at the operating system level makes secure app development much easier and encourages consistency among different apps.

2.5.2 Who is the real author of a given app?

In order to communicate to the user the fact that she is interacting with a certain app, we need to turn its unique identifier (the package name, as explained in Section 2.1) into a message suitable for screen presentation. This message must also provide sufficient information for the user to decide whether to trust it with sensitive information or not.

To this aim, we decided to show to the user the app's developer name and to rely on the Extended-Validation [27] HTTPS infrastructure to validate it, since Extended-Validation represents the current best-practice solution used by critical business entities (such as banks offering online services) to be safely identified by their users. As we will discuss in the following paragraphs, other solutions could be used, but they are either unpractical or unsafe.

As a first example, the most obvious solution to identify an application would be to show the app's name as it appears in the market, but we would need to rely on the market to enforce uniqueness and trustworthiness of the names, something that the current Android markets do not readily provide. The existence of multiple official and unofficial markets and the possibility of installing apps via an *apk* archive (completely bypassing the markets and their possible security checks), make this a complex task. In fact, we observed several cases in which apps mimic the name and the icon of other apps, even in the official Google Play market: as an example, Figure 2.4 shows how a search for the popular "2048" game returns dozens of apps with very similar names and icons. For this reason, establishing a root of trust to app names and icons (such as in [12]) is fundamentally unreliable, as these are easily spoofed, even on the official market.

The only known type of vetting on the Google Play market involves a staff-selected app collection represented on the market with the "Top Developer" badge [28]. This is, to our knowledge, the only case where market-provided names can be reasonably trusted. Unfortunately, this validation is currently performed on a limited amount of developers. Moreover, no public API exists to retrieve this information. When an official method to automatically and securely obtain this information is released, our system could be easily adapted to show names retrieved from the market for certified developers, automatically protecting many well-known apps.

Relying on market operators is not, however, the only possible solution. The existing HTTPS infrastructure can be easily used for the same effect. This system also allows users to transfer their training from the browser to the mobile world: using this scheme, the same name will be displayed for their bank, for instance, whether they use an Android app or a traditional web browser.

As far as identifying the developer to the user, two main choices are possible in the current HTTPS ecosystem. The first one simply associates apps with domain names. We need to point out, however, that domain names are not specifically designed to resist spoofing and the lack of an official vetting process can be troublesome.

On the other hand, Extended-Validation (EV) certificates are provided only to legally-established names (e.g., "PayPal, Inc."), relying on existing legal mechanisms to protect against would-be fraudsters, thus preventing a malicious developer to use a name mimicking the one of another (e.g., using the name "Facebuuk" instead of "Facebook"). Extended-Validation certificate are the current mechanism in use by web browsers to safely identify the owner of a domain and they are available for less than $150 per year: in general, a substantially lower cost than the one involved in developing and maintaining any non-trivial application.

Concretely, to re-use a suitable HTTPS EV certification with our protection mechanism, the developer simply needs to provide a domain name (e.g., `example.com`) in a new specific field in the app's manifest file, and make a `/app_signers.txt` file available on the website containing the authorized public keys. During installation (and periodically, to check for revocations), this file will be checked to ensure that the developer who signed the app[7] is indeed associated with the organization that controls `example.com`. If desired, developers can also "pin" the site certificate in the app's manifest.

[7]Recall that all *apk* archives must contain a valid developer signature, whose public key must match the one used to sign the previous version during app updates.

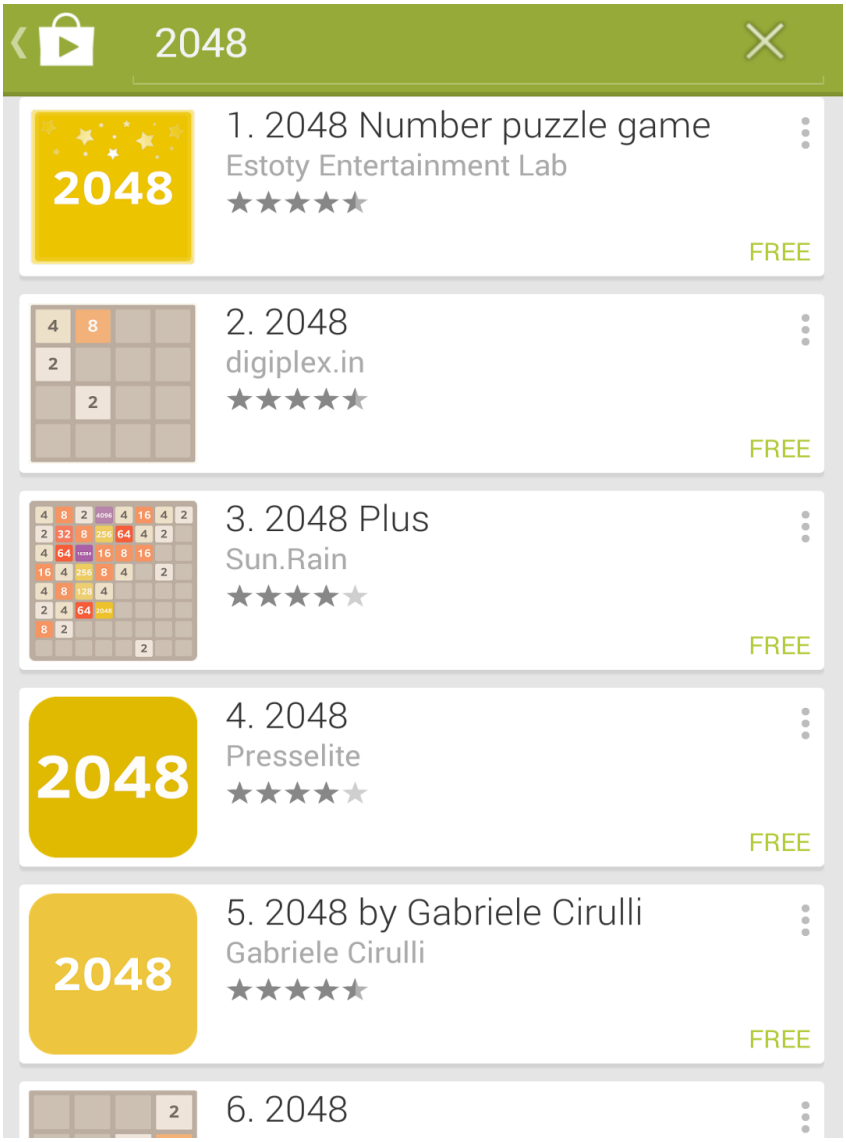


Figure 2.4: A search for the popular "2048" game, returning several "clones." The app developed by the inventor of the game is listed in fifth position.

It should be noted that several issues have been raised on the overall structure of the PKI and HTTPS infrastructure (for a summary see, for instance, [29]). Our defense does not specifically depend on it: in fact, it should be kept in line with the best practices in how secure sites and browsers interact.

2.5.3 Conveying trust information to the user

The two components we have described so far determine the possible statuses of the screen, summarized in the first two columns of Table 2.7. The three right columns of Table 2.7 present our choices, modeled after the user knowledge, training, and habit obtained through web browsers, since the mobile environment shares with them important characteristics:

- The main content can be untrusted and interaction with it can be unsafe.
- It is possible for untrusted content to purport to be from reputable sources and request sensitive user information.
- Cross-entity communications must be restricted and controlled appropriately.

Browsers convey trust-related information to the user mainly via the URL bar. Details vary among implementations, but it is generally a user element that is always visible (except when the user or an authorized page requests a fullscreen view) and that shows the main "trusted" information on the current tab.

For a web site, the main trust information is the base domain name and whether the page shown can actually be trusted to be from that domain (determined by the usage of HTTPS, and shown by a "closed lock" icon). A different element is shown when "mixed" trusted-untrusted information is present. Also, the user is warned that an attack may be in effect if the validation fails.

Most importantly, information presented in the URL bar is directly connected to the page it refers to (pages cannot directly draw on the URL bar, nor can they cause the browser to switch to another tab without also changing information shown on the URL bar).

On the Android platform, we choose the navigation bar as the "trusted" position that will behave like the URL bar. As browsers display different URL bars for different tabs, we also dynamically change information shown on the navigation bar: at every instant in time, we make sure it matches the currently visible status (e.g., the bar changes as Activities are moved on top of the stack, no matter how the transition was triggered). In other words, the security indicators are always shown as long as the navigation bar is.

The navigation bar is in many ways a natural choice as a "trusted" GUI in the Android interface, as apps cannot directly modify its appearance and its functionality is vital to ensure correct user interaction with the system (e.g., the ability for a user to go back to the "home" page or close an app).

Fullscreen apps. To ensure our defense reliability and visibility, our defense mechanism needs to deal with scenarios in which an application hides the content of the navigation bar (on which we show our security indicator) by showing a fullscreen Activity. This allows a malicious application to render a fake navigation bar in place of the original one.

For this reason, to further prove the authenticity of the information shown by our defense system, we complemented our system by using a "secret image" (also called security companion). This image is chosen by the user among a hundred different possibilities (images designed to be recognizable at a small size) and it is displayed together with our lock indicator (see Figure 2.1) making it impossible to correctly spoof it. In fact, a malicious application has no way to know which is the secret image selected by the user.

This system is similar to the "SiteKey" or "Sign-in Seal" mechanisms used by several websites to protect their login pages (i.e., [10], [11]), with the considerable advantage that users are constantly exposed to the same security companion whenever they interact with verified apps or with the base system.

The user has the opportunity to select the secret image during the device's first-boot or by using a dedicated system application. After that a secret image is selected, its functionality is briefly explained to the user. To prevent a malicious application from inferring the image chosen by the user, we store it in a location unreadable by non-system applications.

In addition, we modify the system so that the chosen image will not appear in screenshots (note that the Android screenshot functionality is mediated by the operating system). Also note that non-system applications cannot automatically take screenshots without explicit user collaboration.

We also propose the introduction of a fullscreen mode which still shows security indicators (but not the rest of the navigation bar), in case apps designed for fullscreen operation wish to show their credentials on some of their Activities.

Finally, we prevent applications from creating "inescapable" fullscreen Windows, by simply removing the possibility to use the specific Window's type that makes it possible (refer to Section 2.3.2 for the technical details). As pointed out in Section 2.4.2, we do not expect this change in the current Android API to interfere with any existing benign application.

2.5.4 Implementation

Our prototype is based on the Android Open Source Project (AOSP) version of Android (tag *android-4.4_r1.2*). Some components are implemented from scratch, others as modifications of existing system Services.

The proposed modifications can be easily incorporated into every modern Android version, since they are built on top of standard, already existing, user-space Android components. Their footprint is around 600 LOCs, and we ported them from Android 4.2 to 4.4 without significant changes.

Interaction-target app detection. This component retrieves the current state of the Activity stack and identifies the top app, by accessing information about the Activity stack (stored in the ActivityManager Service).

We also check (via the WindowManager Service) if each Window currently drawn on the device respects at least one of the following three properties:

1. The Window has been generated by a system app.
2. The Window has been generated by the top app.
3. The Window has not been created with flags that assign it a Z-order higher than that of the top-activity Window.

If all the drawn Windows satisfy this requirement, we can be sure that user interaction can only happen with the top app or with trusted system components. This distinguishes the second and third row of Table 2.7.

Database and author verification Service. A constantly-active system Service stores information about the currently installed apps that purport to be associated with a domain name. This Service authenticates the other components described in this section and securely responds to requests from them.

This Service also performs the HTTPS-based author verification as described previously[8]. The PackageManager system Service notifies this component whenever a new app is installed.

[8]For our evaluation prototype, static trust information was used to demonstrate attacks and defense on popular apps without requiring cooperation from their developers.

User interaction modification. The navigation bar behavior is modified to dynamically show information about the Activity with which the user is interacting, as described in Table 2.7. We also added a check in the ActivityManager Service to block apps from starting when necessary (cases listed in the fourth and fifth rows of Table 2.7).

2.6 Evaluation

We performed an experiment to evaluate:

- The effectiveness of GUI confusion attacks: do users notice any difference or glitch when a malicious app performs a GUI confusion attack?
- How helpful our proposed defense mechanism is in making the users aware that the top Activity spawned by the attack is not the original one.

We recruited human subjects via Amazon Mechanical Turk[9], a crowd-sourced Internet service that allows for hiring humans to perform computer-based tasks. We chose it to get wide, diversified subjects. Previous research has shown that it can be used effectively for performing surveys in research [30]. IRB approval was obtained by our institution.

We divided the test subjects into three groups. Subjects in *Group 1* used an unmodified Android system, to assess how effective GUI confusion attacks are on stock Android. Subjects in *Group 2* had our on-device defense active, but were not given any additional explanation of how it works, or any hint that their mobile device would be under attack. This second group is meant to assess the behavior of "normal" users who just begin using the defense system, without any additional training. To avoid influencing subjects of the first two groups, we advertised the test as a generic Android "performance test" without mentioning security implications. Finally, subjects in *Group 3*, in addition to using a

[9] https://www.mturk.com

system with our on-device defense, were also given an explanation of how it works and the indication that there might be attacks during the test. This last group is meant to show how "power users" perform when given a short training on the purpose of our defense.

Subjects interacted through their browser[10] with a hardware-accelerated emulated Android 4.4 system, mimicking a Nexus 4 device. For subjects in Group 2 and Group 3, we used a modified Android version in which the defense mechanisms explained in Section 2.5 had been implemented.

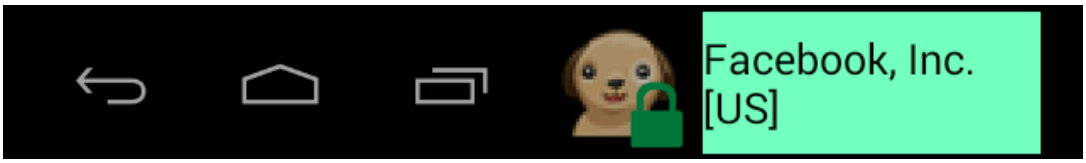
2.6.1 Experiment procedure

The test starts with two general questions, asking the subjects i) their age and ii) if they own an Android device. These questions are repeated, in a different wording, at the end of the test. We use these questions to filter out subjects that are just answering randomly (once given, each answer is final and cannot be reviewed or modified).

Then, subjects in Group 2 and Group 3 are asked to choose their "security companion" in the emulator (which is, for example, the image of the dog in Figure 2.1), picking among several choices of images as they would be asked to do at the device's first boot to set up our defense. The selected image will be then shown in our defense widget on the navigation bar.

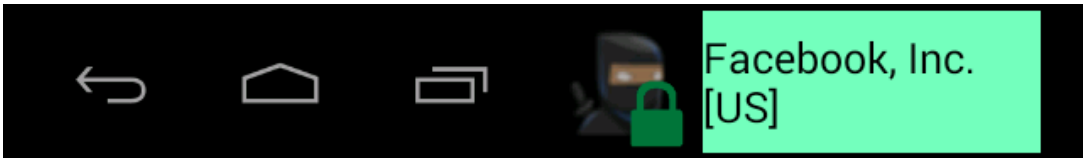
Then, subjects are instructed to open the Facebook app in the emulator. We chose this particular app because it is currently the second most popular free app, and it asks for credentials to access sensitive information. The survey explains to our subjects that the screen of a real Nexus 4 device is being streamed to their browser, and that the application they just opened is the real one. We have included this step because, in a

[10] We used the noVNC client, http://kanaka.github.io/noVNC



Task B_1 and Task B_2 (real Facebook app)

Task A_{std} (non-fullscreen attack app)



Task A_{full} (fullscreen, defense-aware, attack app)

Figure 2.5: Appearance of the navigation bar for subjects using our defense (Group 2 and Group 3), assuming they chose the dog as their security companion. Note that a non-fullscreen app *cannot* control the navigation bar: only a fullscreen app can try to spoof it. In all attacks, the malicious application was pixel-perfect identical to the real Facebook app.

previous run of our experiment, a sizable amount of our subjects did not believe that the phone was "real," and so they did not considered as "legitimate" *any* interaction they had with it.

Subjects are then instructed to open the Facebook app in the emulator several times, leaving them free to log in if they want to. After a few seconds, we hide the emulator and ask our subjects about their interaction. Specifically, we ask if they think they interacted with the original Facebook application as they did at the very beginning. Subjects had to respond both in a closed yes-no form and by providing a textual explanation. We used the closed answers to quantitatively evaluate the subjects' answers and the open ones to get insights about subjects' reasoning process and to spot problems they may have had with our infrastructure.

Table 2.8: Results of the experiment with Amazon Turk users.
Percentages are computed with respect to the number of *Valid Subjects*.

	Group 1: Stock Android	**Group 2**: Defense active. Subjects not aware of the possibility of attacks	**Group 3**: Defense active, briefly explained. Subjects aware of the possibility of attacks
Total Subjects	113	102	132
Valid Subjects	99	93	116
Subjects answering correctly to Tasks:			
B_1 and B_2	67 (67.68%)	70 (75.27%)	85 (73.28%)
A_{std}	19 (19.19%)	60 (64.52%)	80 (68.97%)
A_{full}	17 (17.17%)	71 (76.34%)	86 (74.14%)
A_{std} and A_{full}	8 (8.08%)	55 (59.14%)	67 (57.76%)
A_{std} and B_1 and B_2	4 (4.04%)	51 (54.84%)	73 (62.93%)
A_{full} and B_1 and B_2	6 (6.06%)	63 (67.74%)	76 (65.52%)
A_{std} and A_{full} and B_1 and B_2	2 (2.02%)	50 (53.76%)	66 (56.90%)

We decided against evaluating the effectiveness of our defense by checking if users have logged in. This is because, in previous experiments, we noticed that security-conscious users would avoid surrendering their personal credentials in an online survey (regardless of any security indicator), but would not be careful if provided with fake credentials. Instead, we decided to ask the subjects to perform four different tasks: B_1, B_2, A_{std}, and A_{full}.

During Task B_1 and Task B_2, subjects are directed to open the Facebook app. In these two tasks, this will simply result in opening the real Facebook app.

In Task A_{std} we deliver the attack described in Section 2.2.3 while the subjects are opening Facebook. As a result, the device will still open the real Facebook app, but on top of it there will be an Activity that (even though it looks just like the real Facebook login screen) actually belongs to our malicious app. In Groups 2 and Group 3, which have our defense active, our widget in the navigation bar will show that the running app is not certified, by showing no security indicator on the navigation bar. Therefore, subjects in Group 2 and 3 may detect the attack by noticing the missing widget.

Differently, in Task A_{full}, we simulate a fullscreen attack. In this case, our malicious app will take control of the whole screen. The malicious app can mimic perfectly the look and feel of anything that would be shown on the screen, but it cannot display the correct security companion (because it does not know which one it is). The fullscreen attack app must then mimic to its best the look of our defense widget, but it will show a different security companion, hoping that the user will not notice. For this reason, subjects in Group 2 and Group 3 can detect the attack if (and only if) they notice that our widget is not showing the "correct" security companion they had chosen. Note that this puts our defense in its worst-case scenario, with pixel-perfect reproduction of the original app and the defense widget except for the user-selected secret image.

Note that for subjects in Group 1 this task looks exactly the same as Task A_{std}: if the navigation bar never shows security indicators, we assume it would be counterproductive for an attacker to drastically alter it by showing a "spoofed" security indicator.

The four tasks are presented in a randomized order. This prevents biasing results in case performing a task during a specific step of the experiment (e.g., at the beginning) could "train" subjects to answer better in subsequent tasks.

Figure 2.6.1 summarizes what has been shown on the navigation bar to the subjects in Group 2 and Group 3 during the execution of the different tasks.

2.6.2 Results

In total, 347 subjects performed and finished our test. However, we removed 39 subjects because the control questions were inconsistent (e.g., How old are you? More than 40. What's your age? 21.), the same person tried to retake the test, or the subject encountered technical problems during the test. This left us with 308 valid subjects in total. The results of the experiment are shown in Table 2.8.

The vast majority of subjects in Group 1, using stock Android, were not able to correctly identify attacks and often noticed no difference (typically, answering that they were using the real Facebook in all tasks) or reported minimal animation differences due to the reduced frame rate and emulator speed (unrelated to the attacks). This corroborates our opinion that these attacks are extremely difficult to identify. In particular, only 8.08% of the subjects detected both attacks and only 2.02% of the subjects answered all questions correctly. Manual review of the textual answers revealed that this happened randomly (that is, the subjects did not notice any relevant graphical difference among the different tasks).

Comparing results for Group 1 and Group 2, it is clear that the defense helped subjects in detecting the attacks. Specifically, the percentage of correct detections increased from 19.19% to 64.52% for Task A_{std} ($\chi^2 = 40.68$, $p < 0.0001$)[11] and from 17.17% to 76.34% ($\chi^2 = 67.63$, $p < 0.0001$) for Task A_{full}. Also, the number of subjects able to answer correctly all times increased from 2.02% to 53.76% ($p < 0.0001$, applying Fisher's exact test).

Comparing detection results of the two attacks, we found that the detection rate for the fullscreen attack is slightly better than the one for the non-fullscreen one. However, this difference is not statistically significant. In particular, considering Group 2 and Group 3 together, 66.99% of the subjects answered correctly during Task A_{std} and 75.12% answered correctly during Task A_{full} ($\chi^2 = 3.36$, $p = 0.0668$).

We also noticed that the number of subjects answering correctly during the non-attack tasks (Tasks B_1 and B_2) did not increase when our defense was active. In other words, we did not find any statistical evidence that our defense leads to false positives.

Finally, results for Group 2 and Group 3 are generally very similar, with just a slight (not statistically significant) improvement for subjects in Group 3 in the ability to answer correctly all questions ($\chi^2 = 0.21$, $p = 0.6506$). This may hint to the fact that our additional explanation was not very effective, or simply to how the mere introduction of a security companion and defense widget puts users "on guard," even without specific warnings.

[11] We evaluate results using 95% confidence intervals. Applying the Bonferroni correction, this means that the null hypothesis is rejected if $p < 0.01$.

2.6.3 Limitations

As mentioned, we took precaution not to influence users' choices during the experiment. In particular, subjects in Group 2 used a system with our defense in place, but without receiving any training about it before. Nonetheless, they had to set up their security companion prior to starting the experiment, as this step is integral to our defense and cannot be skipped when acquiring a new device. We designed our experiment to simulate, as accurately as possible, the first-use scenario of a device where our proposed defense is in place. In this scenario, users would be prompted to choose a security companion during the device's first boot. We acknowledge, however, that this step may have increased the alertness of our subjects so that our results may not be completely representative of the effect that our defense widget has on users, especially over a long period of time.

Similarly, the fact that subjects, at the beginning of the experiment, were made to interact with the original Facebook application may have helped them in answering to the different tasks. However, we assume it is unlikely that users are being attacked by a malicious app performing a GUI confusion attack during the very first usage of their device.

It is also possible that the usage of an emulator, accessed using a web browser, may have had a negative impact on the subjects' ability to detect our attacks. It should be noted, however, that the usage of an x86 hardware-accelerated emulator (and VNC) resulted in a good-performance, to the point we would recommend this setup to future experimenters (unless, of course, they have the time and resources to gather enough participants and use real devices).

Finally, there is a possibility that the subject's network was introducing delays. From the network's point of view, the emulation appears as a continuous VNC session from the beginning to the end. This setup should not specifically affect individual tasks, but may have caused some jitter for subjects.

2.7 Conclusions

In this work, we analyzed in detail the many ways in which Android users can be confused into misidentifying an app. We categorized known attacks, and disclose novel ones, that can be used to confuse the user's perception and mount stealthy phishing and privacy-invading attacks.

We have developed a tool to study how the main Android GUI APIs can be used to mount such an attack, performing a full state exploration of the parameters of these APIs, and detecting problematic cases.

Moreover, we developed a two-layered defense. To prevent such attacks at the market level, we have developed another tool that uses static analysis to identify code in apps that could be leveraged to launch GUI confusion attacks, and we have evaluated its effectiveness by analyzing both malicious applications and popular benign ones.

To address the underlying user interface limitations, we have presented an on-device defense system designed to improve the ability of users to judge the impact of their actions, while maintaining full app functionality. Using analogies with how web browsers present page security information, we associate reliable author names to apps and present them in a familiar way.

Finally, we have performed a user study demonstrating that our on-device defense improves the ability of users to notice attacks.

Chapter 3

Trusting the Operating System for Remote Authentication

Mobile applications ("apps") have evolved from being simple conveniences, into complex systems, aimed at powering the latest generation of Internet-connected, distributed, massively multi-user services. This implies that these apps depend to some extent on backend services to function. For example, many apps function as frontends for existing online services, where their entire behavior is tightly coupled to the remote service. To handle multiple users securely on these backends, some sort of authentication needs to occur.

Traditionally, this procedure relies on a combination of "user-private" credentials, such as username and password. However, given the incredibly crowded market in which these apps compete and the fickle nature of users, there is a significant pressure to lower the "friction" new users encounter when using an app. For this reason, applications are moving away from authentication schemes based on user-private credentials, toward those schemes that are more automatic. An existing solution that is often used to accomplish this is OAuth, an authorization mechanism that can enable users to leverage accounts

on identity services, such as Google and Facebook, without creating new, ad-hoc, ones. Nonetheless, developers constantly strive to create novel, custom authentication mechanisms to increase the ease-of-use of their applications.

In this paper, we study and characterize a new broad class of vulnerable authentication schemes, which fully rely on what we call *device-public* information. With this term, we refer to all information, properties, and data that can be accessed by any application (with proper permissions, as explained in Section 3.2.2) installed on the same device.

As an example, consider a messaging app that, after users identify themselves, stores a token in the device's external storage, which any app can access. The app then sends this token to the app's backend server each time it is used, as a form of authentication. This technique has the advantage that if the user uninstalls the app and later wants to use it again, the token will persist on the external storage, and no re-authentication will be required. Unfortunately, this versatility comes at a price: a malicious app running on the same device can obtain and leak this token to a remote attacker, who can now easily hijack the user's account. Even if an app is leveraging a technology such as OAuth, poor handling of the resulting tokens could render them device-public as well. This is just one possible scenario for the mis-use of device-public information; apps can and do use such schemes as the *only* form of authentication, without requiring private data from the user such as a password, rendering their associated accounts wide-open to malicious apps on the device.

In work presented in this chapter, we perform the first comprehensive analysis and characterization of vulnerable authentication schemes based on device-public information. We start by describing the *identity-transfer* attack, a generic exploitation technique, composed by two steps. First, a malicious app, termed the "ID Leaker," steals all device-public information from a victim's device without any user interaction. Then, this app transfers this data to "ID Injector," an app installed on the attacker's device that collects

the received information and injects them into the device. Once this step is completed, the attacker can simply install the vulnerable target app, which will automatically log the attacker in the victim's account.

We also take the first step toward understanding how widespread this class of vulnerabilities is, by developing a dynamic analysis system that aims at uncovering potentially-vulnerable apps among a much larger set. While "authentication" is a difficult behavior to characterize, we can leverage interesting behavioral patterns to locate authentication with enough accuracy to help a human analyst determine if a vulnerability is present. In particular, the system we developed records the app's user interface behavior during its first execution on a device, when authentication and registration is likely to appear. Then, as a second step, the system wipes the app's private data (by uninstalling and then re-installing the app), and it runs the app once again. The key intuition is that if the behavior of an app changes after the re-installation, it means the app somehow relies on device-public information for authentication, and is very likely to be vulnerable to our attack. As a final step, our system attempts to confirm the vulnerability by using the generic exploitation technique described above to transfer the identity used in the previous steps to an entirely new device.

Although some of the ideas and intuitions behind this work can be applied to any mobile operating system (and the corresponding apps), in this work we focus on Android. This choice has been motivated by two main reasons: the fact that Android is currently the most widespread mobile operating system [31] and the ease of performing automatic analyses on Android apps.

We used this analysis system to vet 1,000 of the most popular applications from the Google Play Store, and 41 of them were correctly identified as vulnerable. Two of these vulnerable apps were WhatsApp and Viber, two of the most popular messaging apps, which are used by hundreds of millions of users. For both these apps, we discovered that

it was sufficient to *steal* the content of a single file (and spoof the value of some device's identifiers) to fully hijack a user account. We reported our findings to the respective security teams, which quickly acknowledged the vulnerabilities. Among the apps flagged as vulnerable, our system also identified several popular games that allow a user to purchase virtual objects or currency: our automatically generated exploit was able to hijack these accounts as well. We conclude this work by proposing and implementing solutions for the identified class of vulnerabilities.

In summary, the contributions of this work are as follows:

- We identify and study a new class of insecure authentication schemes that rely on device-public information.
- We demonstrate how it is possible to automatically exploit these vulnerable schemes by developing a generic "identity-transfer" attack, which is capable of stealing and replaying device-public information to hijack accounts.
- We explore the scope of the vulnerability in 1,000 popular apps from the Google Play Store using an automated dynamic analysis system, and identified 41 vulnerable apps, including Viber and WhatsApp.
- We propose and implement solutions to the identified problems.

3.1 Authentication Schemes

Authentication in mobile applications can take on a variety of distinct forms, with differing security properties. The first, and most obvious, authentication scheme is the traditional username and password, in which the user is asked directly by the authenticating app for credentials. The app then sends these credentials to its backend server, which

verifies their correctness. After this step, the server sends to the app a *token*, which is a shared secret string that can be used for authenticating all following interactions between the client and the server.

Another way to authenticate is to use third-party authentication services. This method removes the need to handle tedious per-app registrations. In Android, the AccountManager [32] offers a generic API that can be used to obtain an OAuth-like authentication token from third-party identity providers, such as Google or Facebook. The obtained token is presented with the app's requests to its backend, and can then be used by the backend to ask the third-party service for more information about the user.

Another popular scheme uses text messages (SMS) and the user's phone number as a form of authentication. In this scenario, the user would need to prove that they own a given phone number. As a part of the verification process, the user would typically enter the phone number manually. A code is then sent via SMS to the user, and is typically parsed automatically from the user's SMS inbox and verified. After this step, the phone number is used as the user's primary identity.

Lastly, some Android apps employ schemes in which distinguishing information about the device itself is used to bind a device to an account. This works under the implicit assumption that these identifiers are static and unique per device. To authenticate, the required identifiers are sent to the app's backend server, an authentication token is obtained, and such token is then sent along with future requests.

To reach the widest possible audience, many apps offer multiple authentication schemes, such as Facebook, Google, or regular user name and password authentication. While some of these methods may be securely implemented, the app may still be vulnerable if it allows users to use unsafe login-less methods that rely only on device-public information.

At their core, all these authentication schemes aim to obtain some sort of *token* that can be used to authenticate a user to the app's remote backend. However, if the authentication token can be obtained using information that another application on the same device can obtain, the authorization scheme is not safe. We also note that, even when apps employ schemes that are thought to be secure, they can still be vulnerable to account hijacking if they store authentication tokens in publicly accessible locations.

3.2 Identity-Transfer Attack

Our key observation is that if an app only relies on device-public information to authenticate the user to its backend, it is possible for a malicious app to mine and leak all relevant information. If such a scheme is in use, an attacker can perform an identity-transfer attack, transferring information from the victim's device to the attacker's, so that the user's identity associated to a given app is effectively transferred.

3.2.1 Threat Model

In this chapter, we assume that an attacker is able to lure the user into installing an attacker-controlled malicious application. This application requests all the needed permissions to acquire the device-public information being stolen, as outlined in Section 3.2.2. Moreover, we assume the operating system of the device to be uncompromised, and it thus constitutes a trusted computing base. Furthermore, we assume that the victim's device is *not rooted* (if it is, our attack does not take advantage of it), which means that an attacker cannot get *root* privileges. Therefore, the malicious app does not have access to app-private data, as the separation of the apps' private storage is strictly enforced by the OS.

Table 3.1: Considered sources of *device-public* information.

Source	Required Permission	Survives factory reset	Linked to a Google Account	Linked to a SIM Card
ANDROID_ID	–	–	–	–
IMEI	READ_PHONE_STATE	✓	–	–
WiFi MAC address	ACCESS_WIFI_STATE	✓	–	–
Bluetooth MAC address	BLUETOOTH	✓	–	–
Google account email	GET_ACCOUNTS	–	✓	–
Google Service Framework ID	READ_GSERVICES	–	✓	–
Google Advertising ID	–	–	✓	–
Phone Number	READ_PHONE_STATE or READ_CALL_LOG	–	–	✓
Incoming Phone Calls	READ_CALL_LOG	–	–	✓
SIM Card Serial Number	READ_PHONE_STATE	–	–	✓
Received SMS Messages	RECEIVE_SMS or READ_SMS	–	–	✓
External Storage	READ_EXTERNAL_STORAGE	–	–	–

3.2.2 *Device-Public* Information Sources

We refer to *device-public* information as information that can be accessed from any app on the device that requests the permissions needed to obtain it. We will focus primarily on Android versions ranging from Android 4.4 to Android 7. When necessary, we will describe differences among different versions.

Here we will discuss the different sources of device-public information we have considered in our study, which are used by apps to identify users (also summarized in Table 3.1). Some of these identifiers are related to a specific hardware device, and cannot be changed by the user, whereas others can be changed after a "factory reset" of the device, or are linked to a Google Account. Google has recently attempted to hide some identifiers from apps to thwart tracking. That said, as of Android 7.1.1, we found that we are still able to access every identifier mentioned here, save for the Bluetooth MAC address.

Apps may need specific permissions to access some of these sources of device-public information, therefore a careful user may be able to notice that a malicious application is accessing some device-public information. However, while Android 6 introduced a feature alerting the user at the time some permissions are used, a malicious app can bypass this alert by lowering its own "Target SDK Version." By doing this, the old permission model, in which the user is not informed the moment an app uses a permission, is used.

ANDROID_ID. This is a device's unique ID number, set by Android upon a device's first boot or factory reset.

IMEI. The IMEI is a hardware identifier given to each piece of cellular equipment, including the baseband radios of mobile phones.

WiFi MAC address. Similar to the IMEI, MAC addresses are uniquely assigned to most conventional network hardware. The WiFi MAC address can be obtained by any app requesting the `ACCESS_WIFI_STATE`, using the API `WifiInfo.getMacAddress()`. In

Android 6 (and later versions), the behavior of this API has been changed, so that it always returns the value `02:00:00:00:00:00`. However, we found that it is still possible to access this identifier using the `NetworkInterface.getHardwareAddress()` API.

Bluetooth MAC Address. The device's Bluetooth MAC address is a persistent hardware identifier that can be queried by using the API `BluetoothAdapter.getAddress()`. This API requires the `BLUETOOTH` permission. Starting from Android 6, the behavior of this API has been changed, so that it always returns the value `02:00:00:00:00:00`.

ADB serial number. The device's ADB Serial Number, which is used to identify devices on the Android Debug Bridge, is an identifier that persists across factory resets. It can be accessed by querying the `android.os.SystemProperties` object using the key `ro.serialno`.

Google account email. Many Android devices use Google account emails as a form of Single Sign-On, and the email address used can be easily obtained using the AccountManager API.

Google Service Framework ID. This ID is used to identify a user when accessing Google Service Framework applications.

Google Advertising ID. In an attempt to allow users to opt-out of mobile ad tracking campaigns, Google created a specific persistent identifier [33] to be used with advertising. It can be queried by any app (through the `AdvertisingIdClient` class in the Google Play Services), but, unlike the other identifiers, also freely reset by the user. Google's policy [34] states that all advertising must use exclusively this identifier for tracking ("in place of any other device identifiers for any advertising purposes"), although in practice it is often not used [35].

Phone number. We consider the phone number associated with the SIM card inserted in a device as device-public information. A specific API (*getLine1Number*, requiring the *READ_PHONE_STATE* permission) exists to retrieve this value, however the re-

turned value is not always reliable, depending on the SIM Card manufacturer. Various workarounds do exist, including reading the call log, which requires the `READ_CALL_LOG` permission.

Received SMS Messages. Any app (with proper permissions) can request to be notified of the origin and content of new SMS messages.

Incoming Phone Calls. Apps can request to be notified about the basic data of incoming calls, including the caller's number. Additionally, Apps can also read the call history. Interestingly, phone calls can be used for authentication, by using part of the sender's phone number (which remote services can control) as a verification code.

SIM Card Serial Number. In devices where a SIM Card is present, apps can access this identifier, which is tied to the used SIM Card, by using the `getSimSerialNumber` API.

External Storage. Many Android devices today come with, or have the ability to add, some form of external storage, usually in the form of a larger Flash-based storage device or SD card. The precise behavior of external storage differs among Android versions and devices, but, typically, any app can request the `READ_EXTERNAL_STORAGE` permission to access its contents. This gives the app access to the public areas of the external storage, shared by all apps.

Files stored in here are publicly accessible and some of them are not deleted upon app's uninstallation [36]. Therefore, as a usability feature for the users, some apps store authentication cookies in this location, so that the credentials *survive app's re-installation*. Unfortunately, while this may sound a reasonable practice, it is not secure. In fact, in this scenario, an attacker would be able to easily hijack the user's account by reading the files containing these authentication cookies and using their content to authenticate with the victim apps' remote backends.

3.2.3 Proof-of-Concept Attack Implementation

In simple terms, the attack consists of an app on the victim's device, which steals a set of device- and user-specific information, and exfiltrates it to the attacker. The attacker can then inject this information into their own device, so that apps behave seamlessly as if they are still on the victim's device. In particular, the attacker can use vulnerable apps as if authenticated as the victim on the victim's phone.

We implemented the "identity transfer" attack in two different components: the "ID Leaker," and the "ID Injector." The "ID Leaker" app, which could be thought of as a prototypical third-party malicious application, requires the Android permissions to access the SMS, device call notifications, external storage contents, and static device identifiers (refer to Table 3.1). The app then uses the well-documented Android APIs to access and leak the device-public data that constitutes the *user's identity*, and it sends it to the attacker's device. We note that the app's functionality could be easily hidden inside a seemingly legitimate app, and that it can run on completely unmodified devices without requiring any admin privileges (therefore on un-*rooted* devices). We also note that if an attacker aims at hijacking the account of a specific victim app, the "ID Leaker" only requires the permissions needed to access the specific device-public information used by the victim app for authentication purposes.

For the attacker's device, we created the "ID Injector," which takes data from the "ID Leaker" and injects them into an attacker-controlled device. We use the Xposed framework [37] (a tool for performing run-time patching of the Android framework) to easily hook the Android API methods used to query device-public information, and spoof their results to return the data leaked from the victim. The external storage's content is also transferred from the victim and copied into place. Without external information,

there is no way for the app under analysis to tell that the data has been spoofed. Because of our usage of the Xposed framework, the attacker-controlled device (but not the victim's one) must be rooted to properly spoof the received identity.

3.3 Vulnerability Detection

In order to understand how widespread device-public authentication schemes are on Android, we created an automated system to locate vulnerable apps in the wild. This system could also be used by security researchers, software developers, and app market operators to automatically spot weakness in the authentication mechanisms used by the analyzed apps.

While the attack described in Section 3.2.3 is very effective against vulnerable apps, we cannot simply use it against all apps to build a detection system, for two main reasons. First, it is difficult to differentiate success of the attack from other application behaviors, as we have no baseline of the app's normal behavior to compare it to, and cannot link changes in this behavior to device-public information. Second, as we discuss in Section 3.1, "authentication" can be implemented in a variety of ways, making it difficult, if not impossible, to concretely define and locate authentication behaviors in a generalized way. We therefore cannot rely on any direct knowledge of the authentication itself to help understand when our exploit is having an effect.

To address these challenges, we developed an approach that aims at identifying authentication behaviors *indirectly*. We build our approach on the observation that an app behaves differently depending on whether or not it has already authenticated its user to a previously created account, and that this difference will be reflected in the app's user interface.

Thus, as a first step, the system executes the app, provides any requested device-public information to the app, and records the app's behaviors. These behaviors are in the form of a trace of different UI *states* (as detailed in Section 3.3.4). The aim of this initial execution is both to trigger the app's authentication or registration mechanism, as well as to get the server's backend to store some sort of state for the user, which can be observed in future traces.

Next, all app-private information for the app is deleted. We achieve this by uninstalling and re-installing the app. This operation deletes all app's files in private locations.

At this point, the app is executed again and, if the behavior is different from the one observed during the first run, it is possible that the app may be using device-public information (which could be both device's identifiers or publicly accessible files in the external storage) to authenticate the user. Typically, a difference may be observable because of the absence of a "login" screen due to already being authenticated, or the absence of an introductory "welcome" screen due to restoring the previously-saved user state, but more subtle UI modifications are possible.

As a last step, the system confirms the vulnerability, by transferring the device-public information to a different device, executing the app again, and comparing these behaviors with the previous ones. This transfer operation encompasses copying both publicly accessible files and device's identifiers.

In the remainder of this section, we will first discuss in detail the three steps of our analysis, as shown in Figure 3.1. We will then provide several technical details about the underlying dynamic analysis and the comparison of states and traces.

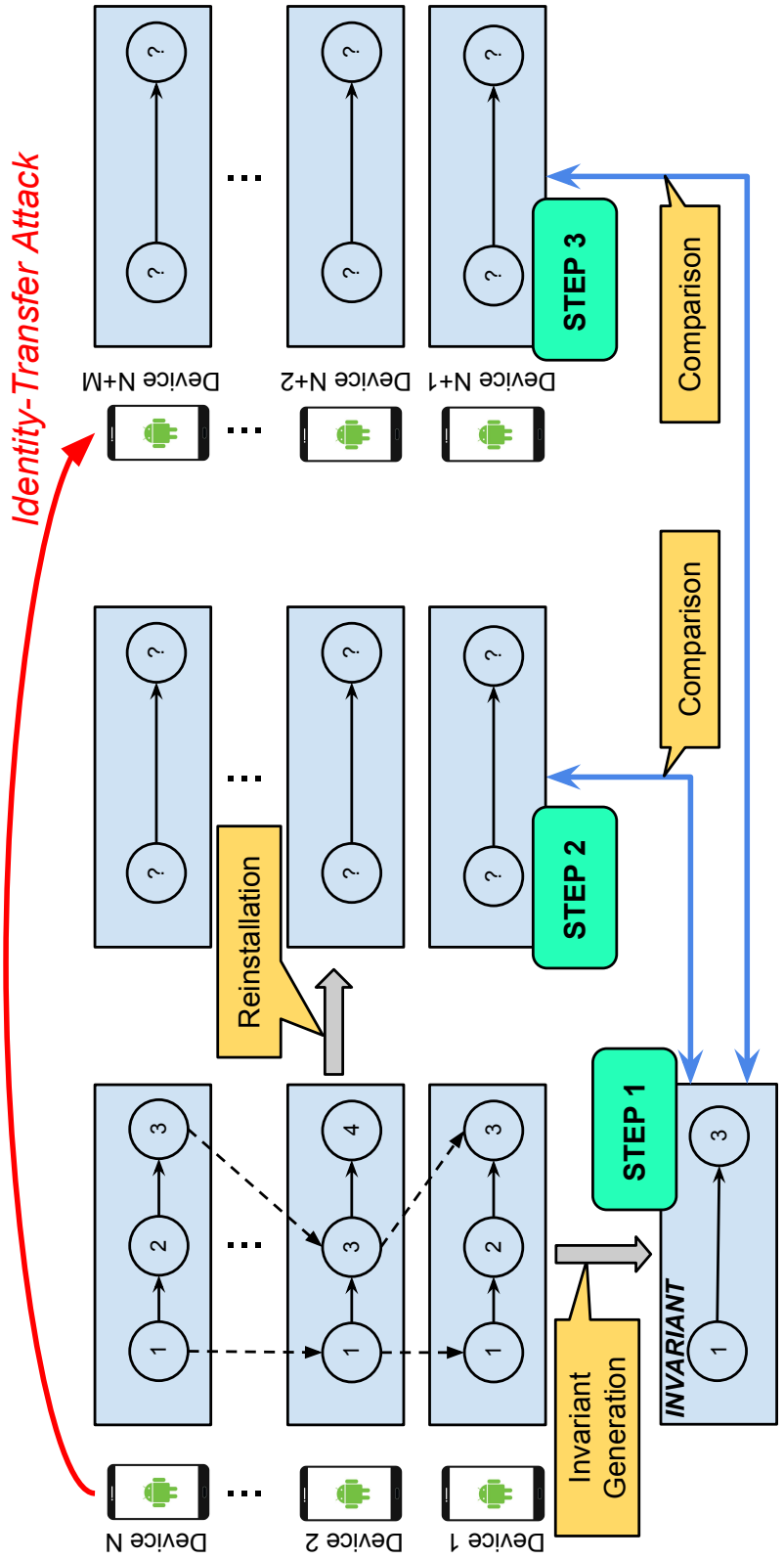


Figure 3.1: Overview of the developed dynamic analysis system.

3.3.1 Step 1: Capturing Initial Behavior

First, we need to characterize the behavior of a given application when installed for the first time on an Android device. Our system functions primarily by collecting and comparing *traces*, consisting of an ordered list of UI *states* encountered during a given execution of the app. Details of how states and traces are collected and compared can be found in Section 3.3.4.

However, our system truly needs to characterize the "normal" behavior, not just merely record one execution. This is far from trivial, mainly due to the fact that dynamic analysis is hindered by non-deterministic behaviors present in apps, the OS, and network communications. To address this challenge, we first execute the analyzed application on multiple devices, collecting multiple traces. Recent work has shown that running the same app multiple times is, in Android, effective in reducing the effects of non-deterministic behaviors during dynamic analysis [38]. The collected app behaviors are then used to compute a so-called *Invariant*, representing the most common set of behaviors. Specifically, the Invariant set is computed as the set of all states that appear in *all* the collected traces.

3.3.2 Step 2: Vulnerability Detection

In this step, we delete all app-private data from the devices used in Step 1, collect new traces, and compare them with the Invariant. We accomplish clearing the app-private data by re-installing the app, which is known to remove all app-private information, including authentication tokens, cookies, databases and other private files.

After re-installation, the app is dynamically stimulated, and traces are collected in the same way as in Step 1. Then, we compare the new traces against the Invariant, looking for behavioral differences in the traces. These discrepancies are typically due to

setup, registration, or login interfaces. Therefore, they are a strong signal the app was able to authenticate with the remote backend, only using information that *survived* the app's re-installation, which must therefore be device-public.

More precisely, if we determine that, during the execution of the analyzed app in this step, at least *one* state present in the Invariant has been skipped in *all* the collected traces, the app is flagged as potentially vulnerable.

3.3.3 Step 3: Exploit Verification

In Step 3, we verify if an app uses an insecure authentication scheme by actually attempting an identity-transfer attack against it.

To perform the attack, we transfer the device-public information stored in the devices used during Step 1 and Step 2 to new devices (which have not been used in the analysis of this app before), as explained in Section 3.2.3. Then, the same procedure used in Step 1 is used to obtain execution traces from the previously-unused devices.

These traces are then compared against the Invariant, as in Step 2. If we detect that at least *one* of the states skipped during Step 2 is also always skipped during Step 3, we conclude that the attack succeeded, and we flag the app as vulnerable.

3.3.4 Dynamic Analysis

In order to accomplish the above steps, we need to deterministically execute an application to trigger the authentication behavior, while minimizing behavioral divergences due to non-deterministic operating system or network behaviors. To this end, our system stimulates apps through their UIs, including buttons, text fields, and other interactive elements, as well as taking note of any incoming SMS and phone calls the used device may receive.

We rely on *uiautomator*[39], both to control the device and to obtain state information about the device itself. We control uiautomator from a normal PC by connecting it to the device using the Android Debug Bridge (ADB) and the uiautomator Python wrapper [40].

Possible actions are derived from the UI's content (button labels, text field descriptions, ...), and inserted into a priority queue. The priorities are arranged such that the most specific actions are performed first. The developed system also keeps track of previously touched UI elements, removing them from the priority list, so that every element is touched at most once. This is done to prevent the stimulation from entering an infinite-loop by continuously interacting with the same element.

The following is a list of the actions that our detection system can perform, in order of priority:

Fill text fields. Our system automatically fills some text fields. In particular, it first determines the type of information a text field is suppose to contain by (similar to [41]) checking labels and IDs associated to each text field against a pre-determined list of strings. Then, if a text field is determined as asking for a phone number, our system fills it with the device's phone number. Likewise, if a text field is determined as asking for a username, our system inserts a randomly generated one. It is important to note that no user-private information (e.g., a password) is inserted during this (or any other) step of the dynamic stimulation of an app.

Touch button. Our system interacts with UI elements that are "clickable." All clickable objects found are prioritized based on their type (e.g., buttons have higher priority than text fields) and their content; this allows us to, for example, touch an "OK" button before a "Cancel" button.

Pseudo-random touch events. If none of the previously mentioned actions can be performed on a state, our system will try to explore the app's behavior by simply randomly clicking on its UI. This situation usually happens, when the application uses custom UI elements, which do not export standard layout information to the OS.

In addition, if the analyzed app loses its focus (e.g., a window is opened in the system browser), we perform appropriate actions to make the analyzed app regain focus.

3.3.5 App States Extraction and Comparison

In order to make meaningful comparisons of different executions, we need a way to collect the current *state* of an app (e.g., which content it is showing to the user) at different times during our analysis and compare those states. The way in which states are encoded and compared needs to be sufficiently informative to capture significant behavioral changes, but also flexible enough to help ignore minor changes unrelated to the app's functionality. Specifically, the behavior of an app is encoded as a trace of states, which are then compared, looking for evidence of vulnerable authentication schemes.

State Extraction. Every five seconds, the system checks if the current device's UI is in a *steady state*. By this, we mean a situation in which the UI is likely not to change if no action is performed. If so, we record the current app's state (as better defined below) and we perform an action. Otherwise, the system waits up to a maximum threshold of 30 seconds. We employ this approach to perform actions and capturing states only when the effects of previous actions on the app's UI are completed. This also allows the sample rate of our system to be dynamic, and it helps to ensure that the captured states make the most sense when compared later. We use information provided by the Android video and input subsystems to know when an animation is being rendered (and

therefore the current state is not steady). However, if we are unable to reach a steady state (e.g., the app uses OpenGL, or is otherwise constantly animating), we resort to an image-comparison approach.

Once the UI is steady, the system records a state, consisting of the following:

- The activity name (in Android, an Activity is a specific UI window)
- A hash of the simplified UI layout data
- A perceptual hash of the device's screen-shot

Hash of simplified UI layout data. To hash the information about the UI elements, we make important simplifications to the UI data, so that it is more easily comparable. In particular, from the layout tree describing the UI state, we remove the information about the location of the different layout's components and the text shown. These positioning or text differences are oftentimes due to intrinsically non-deterministic or rapidly changing UI elements, which are not relevant to our analysis.

Additionally, we take steps to avoid comparing deliberately dynamic content, especially advertising and web content. Advertising on Android is difficult to locate through explicit UI information. However, most mobile advertising is standardized by the International Advertising Bureau [42], which dictates specific pixel dimensions for ads, therefore we filter out elements from the simplified layout that have these sizes. Furthermore, we also filter out all WebView objects, since dynamic web content is typically a significant source of non-determinism. Lastly, we use the MD5 algorithm to condense the state information.

Hash of device's screen-shot. To hash the image acquired during the screen-shot, we use the algorithm called *average hash* provided by the *ImageHash* Python library [43]. This algorithm was chosen to provide meaningful fuzziness for images, abstracting away

small, unimportant differences, such as constantly-animating UI elements. Specifically, this algorithm compresses every image in a 64-bit locality-sensitive fuzzy hash. The algorithm is designed so that images "appearing" as similar for humans are hashed to the same value, regardless of small graphical differences they may have.

State Comparison. We consider two states as equal if all the 3 components described above are equal. Moreover, when comparing states in traces collected during Step 2 against the Invariant, we also consider two states as equal if their image hashes only differ slightly (less than 10% of the bits composing the image hash). This threshold was determined empirically, by taking a subset of the apps from our dataset, and manually determining the optimal value.

Additionally, if during the dynamic stimulation of an app the device receives an SMS or a phone call, we add special states to the trace.

3.4 Experimental Results

3.4.1 Datasets

We used the vulnerability detection system to probe apps from two different datasets:

***"Top Free"* dataset.** A dataset of 606 apps containing all the most popular available free Android apps. To generate this dataset, we first downloaded all the 539 available apps listed in the "Top Free" category on the Google Play market. Then we supplemented this set with other 67 applications starting from the ones that have the highest cumulative number of installations.

***"Top Grossing"* dataset.** A dataset containing the 394 most popular free apps in the "Top Grossing" category on the Google Play market (excluding the ones already present in the "Top Free" dataset). We chose this specific category because experimental results

on the "Top Free" dataset and previous executions of our experiment revealed that apps from this category often allow users to authenticate using non-secure methods to ease their adoption.

Apps from both datasets were downloaded in January 2016. These datasets constitute a heterogeneous corpus of very popular applications both in terms of installations and developers' revenue. In total, we analyzed 1,000 distinct apps.

3.4.2 Experimental Setup

Our system is implemented using a series of Nexus 5 handsets tethered to a controlling PC. Specifically, we used 3 phones during the Invariant Generation and Vulnerability Detection phases (Step 1 and Step 2) and 3 additional phones during Step 3. All handsets run Google's official Android 4.4.4 images (the most adopted Android version at the time the apps were downloaded [44]).

During the collection of every trace of our analysis, we dynamically stimulated an app for two minutes. To ease the deployment of our infrastructure, devices' identifiers and phone numbers were modified during different runs of the experiment, effectively simulating the usage of a new device every time the experiment was run.

Averagely, the experiment needed 458 seconds per app to run Step 1 and Step 2 (including time necessary to reboot a device and install an application). For apps flagged as potentially vulnerable after these two steps, the analysis required, in average, additional 223 seconds per app to run Step 3 (including the time necessary to transfer the device's identity).

3.4.3 Results

Our system flagged 50 apps as vulnerable in our corpus of 1,000 distinct apps. Using manual analysis, we verified that 41 out of the 50 detected apps were actually vulnerable to the identity-transfer attack. Among these, two apps are Viber and WhatsApp, two very popular messaging apps with hundreds of millions of installations. We postpone the discussion of the vulnerabilities identified in these two apps to Section 3.5.1. Another group of 38 apps is composed by popular games, in which an attacker can perform an identity-transfer attack to steal the victim's virtual currency or objects. We will provide more details about them in Section 3.5.2.

Another detected app authenticates users by using standard SMS authentication. Specifically, this app identifies users with their phone number, by sending an authentication code to their phone number using an SMS, which is then automatically read by the app. If this code is stolen, an attacker can login and control all aspects of the user's account.

This security issue is different from the one found in the messaging apps described in Section 3.5.1, as it needs the attacker to steal the content of an SMS received by the victim. However, it still falls into our threat model, since the SMS content is device-public information.

In other 7 detected apps, we were able to transfer an identity with the exploit, but the identity was not protecting anything sensitive. Our system cannot, of course, detect which content is truly sensitive (e.g., related to a user's account) to a particular app, but the differences in the UI were present.

For example, in one app, the device-public information was used to track whether a user had accepted the application's End-User License Agreement (EULA), and, in another one, whether the user configured application preferences. In the other 4 apps, the backend uses this information to track whether a user has viewed certain full-screen "special offers" or advertisement from the app's developer.

In addition, we found an app that, on first usage, shows a sign-in interface, since it assumes that the user does not already have an account. However, on subsequent re-installations, this app shows a username and password login interface, because it infers, using device-public information, that a returning user would already have an account. While not allowing any sort of account compromise, this information can still be leveraged by any other app to infer valuable information about the user, as explored in [45].

Finally, 2 additional apps were detected because of problems of our testing infrastructure, such as connectivity issues of the apps that caused the appearance of different graphical elements between the first installation and the subsequent ones. Subsequent runs of our experiment on these samples confirmed that these apps were detected for temporary problems. We consider these two apps as false positives.

3.5 Case Studies

3.5.1 Messaging Applications

Two of the detected applications are the very popular messaging apps, WhatsApp and Viber, which allow users to send and receive text messages, VOIP calls, and media. While the statistics on the Google Play market are not precise, WhatsApp is estimated to have more than 1 billion installations and Viber has more than 500 million.

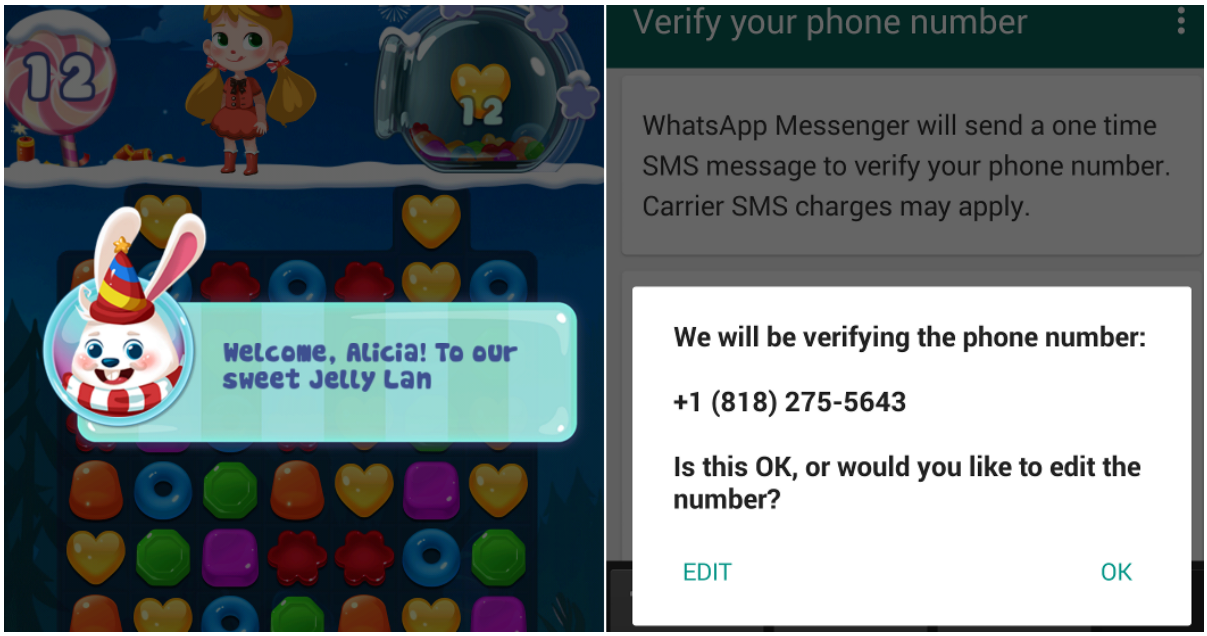


Figure 3.2: Examples of states recorded during the Invariant Generation phase (Step 1). Since these states were not present during Step 2 and Step 3 of our analysis, our system correctly classified these apps as vulnerable. In the left example, the skipped state shows to the user an introductory tutorial of the game. In the right one, the skipped state asks the user to confirm the entered phone number before validating it.

Our vulnerability detection system flagged WhatsApp as vulnerable, since it detected, in the Vulnerability Detection and Exploit Verification phases, the absence of the "confirm your phone number" interface (shown in Figure 3.2), and the missing reception of an incoming SMS, used for authentication. Similarly, while analyzing Viber, the system detected this app as vulnerable because of the missing "Enter Your Name" dialog (shown only to new users) and the missing reception of a phone call whose part of the caller number is used as an authentication token.

Initially, we speculated that those apps were detected as vulnerable because they use the user's phone number, verified using received SMS or incoming phone calls, as their authentication method. This authentication method is common among popular messaging apps, and we consider it as vulnerable in our threat model. In fact, an attacker

with a malicious application installed on the victim's device can pretend to own the victim's phone number and verify it by sending the authentication SMS received (or the caller phone number) from the victim's device to an attacker-controlled device.

However, further analysis surprisingly revealed that an even simpler attack is possible against these apps. The identity-transfer attack was successfully performed for both apps even though their backend did not send any SMS or phone call. This was possible because these apps used the content of a hidden file stored in the external storage to authenticate users upon re-installation.

Therefore, in the version of the apps we have analyzed, an attacker controlling an app on the victim's device could authenticate to the remote backend on behalf of the victim by:

1. Copying the content of a specific file (stored in the external storage of the victim's device) to an attacker-controlled device.

2. In case of WhatsApp, spoof the value of the Google email account. To achieve this, an attacker can use a malicious controlled app on the victim's device to query the AccountManager API and exfiltrate its value to an attacker-controlled device. Then, on the controlled device, the attacker can spoof it by using, for instance, the Xposed framework (as we implemented in the "ID Injector," explained in Section 3.2.3).

3. Open the app.

4. When asked to insert a phone number, specify the victim's one.

After these operations, the vulnerable app running on the attacker's device is automatically logged in as the victim, without even having the victim's device receiving an authentication text message. This exploit gives attackers full use of the victim's account, allowing them to send messages on the victim's behalf, and to receive all future messages sent to the victim.

Vendor Reaction. Upon discovery of these vulnerabilities, we contacted both vendors in August 2015. Both vulnerabilities were quickly acknowledged and, after working with the vendors, mitigations were deployed.

Specifically, both WhatsApp and Viber removed reliance on static identifiers and publicly-accessible files. However, they still rely on the content of a received text message (or, in case of Viber, the caller's number of an authentication phone call) for their primary means of authentication.

Interestingly, after our first notification, Viber was initially changed in a way that was ineffective against our attack. In particular, the file in the external storage remained, but just spoofing it was not enough. We discovered that Viber was changed to also check that other device's identifiers matched the ones used during a previous registration. However, an attacker could just query the values of the different device identifiers using an attacker-controlled app on the victim's device (see Section 3.2.2) and then spoof them on an attacker-controlled device, as implemented in the "ID Injector" (see Section 3.2.3). We note that this attack was working until Viber issued another update, in September 2017, removing reliance on the content of publicly-accessible files to perform authentication.

In addition, after our notification to the vendors (but likely independently from our disclosure), in 2016 both apps implemented new cryptographic measures limiting an attacker's ability to impersonate a user when an account is stolen (via ours or other attacks). In particular, both apps implemented an end-to-end encryption mechanism, based on the usage of a per-user key pair. This functionality allows users to authenticate

and encrypt exchanged messages. For instance, suppose that two users A and B communicate together. This system encrypts the communication channel between A and B using their keys. Moreover, a user, for instance A, can check the value of B's public key and, in case B's public key changes, A would be notified (however, A and B could still communicate together). The same notification would be shown after the aforementioned attack is performed because the per-user key is stored in an *app-private* location, so it cannot be stolen and transferred to the attacker-controlled device.

3.5.2 Free-to-play Games

The other 38 identified vulnerable applications are games, in which an attacker can perform an identity-transfer attack to, steal the victim's virtual currency or in-game objects. In these apps, the system detected, for instance, the fact that graphical interfaces used to enter the user's name, or to show game tutorials and welcome messages were skipped during the Vulnerability Detection and Exploit Verification phases (see Figure 3.2). This indicated that the app authenticated with the remote backend and was able to obtain the user's state.

After an identity-transfer attack was performed, we noticed three different kinds of behaviors on the victim's device, when attacked while the victim is using them. Some of the apps show to the user a generic error message after the attack, such as "Connection Timeout." Others show a message informing the users that "another device" accessed their account. Finally, some of them do not show any information to the victim.

All these games offer in-app purchases, and the virtual currency used is derived from real money, using the Google's In-App billing API (IAB). For this reason, these vulnerabilities are particularly worrisome, since they can represent actual financial loss to the

victims and the apps' developers. One surprising result was that the account transferred during the Exploit Verification phase can include virtual currency purchased through Google's In-App Billing API [46].

Notably, this is not an explicit attack against the In-App Billing API, as explored in previous work [47], but rather that its use in conjunction with the discovered vulnerabilities makes this data vulnerable as well. This is due to how the IAB API is implemented and how it is typically used by developers. In particular, even though the IAB mechanism offers to store information about a user's purchases (in a way which is secure under our threat model), it cannot be easily used as a store of the user's current account balance, since precise accounting is not possible.

Therefore, developers need to store the virtual currency balance differently, in the (potentially unsafe) app's backend. In case of applications vulnerable to identity-transfer attacks, this means the user's paid-for currency is as easy to steal as any other information in the user's account. This is particularly of concern, given the already-established trend in malware on other platforms targeting online game accounts [48].

3.6 Proposed Defenses

We propose two defenses against the attack studied in this work: one aimed at creating secure device identifiers, and another aimed at safeguarding SMS-based authentication. A fully working prototype implementation of our defenses is publicly available [49], as an Xposed framework's module.

3.6.1 Securing the SMS Channel

Design. All installed apps on a device (that request the proper permission) can request to be notified of the content of incoming SMS messages, even when these messages are only intended for use by a particular app. As we shown, this behavior is particularly problematic when received SMS messages contain authentication codes destined for only a particular app.

Our proposed solution, similar to one discussed in [50], works by delivering authentication SMS only to the apps intended to receive them. Specifically, we propose a convention that authentication-related SMS should be pre-pended with the string `AUTHCODE: app_cert_fingerprint`, where `app_cert_fingerprint` is the fingerprint of the certificate used to sign the destination app. The OS would then route the message only to the main SMS reader app, and the app bearing the included fingerprint. This improves on Mulliner et al.'s previous solution by not requiring the OS to be notified ahead of time about how incoming messages should be routed.

For example, consider an app named "Foo Messaging" signed with a certificate whose fingerprint is `0d5af23c`. In this case, users enter their phone number into the app, which is sent to the app's backend. As a response, the app's backend sends an SMS message with the content `AUTHCODE: 0d5af238c Verification Code: 34782`. When received, the OS would then only notify the user's default SMS reader and "Foo Messaging" about the new message.

To improve usability, we propose that the default messaging app, by default, hides this routing information. Alternatively, the default messaging app could replace it with an indication of the app the message was delivered to. This functionality can be implemented without any modification to existing apps. However, it would require a small modification to the app's backends to prepend the app's fingerprint to the outgoing SMS.

We note that the recently-released Android version 8 introduces a new API called *createAppSpecificSmsToken*. This API "creates a single use app specific incoming SMS request for the calling package" [51]. When using this API, an app would first get a secret token from the operating system. Then the app's backend would send an SMS containing that specific token to the user's device and the SMS will be subsequently automatically routed by the OS to the correct app (through the token) and it will not be made readable by any other apps (or visible by the user).

While it may seem that this new feature mitigates the weakness of the usage of SMS to authenticate user, we argue that, on the contrary, it eases the attack we have described in this chapter. In fact, while the usage of this API would stop an attacker from stealing and replaying authentication codes when the user attempts to authenticate, the attacker can just attempt their own authentication (simulating a user re-installing the app on the same device), at which point the SMS will be sent and routed to the attacker's app and it can thus be easily stolen. From the conceptual point of view, the attack works because the app's backend does not have enough information to determine whether the app receiving the SMS is the legitimate app or the attacker's app.

There are two additional aspects that make apps using this new API more vulnerable to the attacks presented. First, neither the user nor the legitimate app will notice the incoming SMS message (triggered by the attacker), since it will be routed only to the attacker's app. Second, the attacker's app does *not* need to require the *READ_SMS* permission when receiving messages using this API, thus making this malicious app stealthier.

Implementation. There are two ways an app can access SMS in Android: an app can ask to the operating system to be notified when a new message is received, or an app can access the list of received messages. Thus, to implement our defense, we modified both the Android *InboundSmsHandler* component, responsible to notify apps of incoming

messages, and the *SmsProvider* component, mediating apps' accesses to received SMS. Globally, our modifications consist of approximately 100 LOC added to the original Android code. The added code introduces an average slowdown of 5.381ms every time an SMS is received and a slowdown of 2.064ms every time an app queries the operating system for received SMS. We consider both slowdowns as negligible, given the fact that, receiving a text message it is not a frequent event.

3.6.2 Secure Device IDs

Design. The most common and easily obtained device-public identifier is the ANDROID_ID, which is intended to be used to allow apps and their backends to differentiate Android devices.

In our defense we modify the API used to access the ANDROID_ID, so that it returns a Private Device ID ($PDID$) different for every app (more precisely, different for every app's signing certificate), instead of the original device-wide value. Specifically, the first time a device boots (or after a factory reset) a random Secret ID (SID) is generated. The Private Device ID is then derived from the Secret ID using the signing certificate included with each app, which uniquely identifies its developer. In this way, the semantics of the ANDROID_ID are preserved, apps from different developers cannot steal each other's identifiers, but no convenience is lost for a developer with multiple apps on the same device. Moreover, the $PDID$ does not change after app's re-installation.

Specifically, the $PDID$ is computed as follows: `HMAC(SID, caller_app_cert_fingerprint)` where: `caller_app_cert_fingerprint` is the certificate fingerprint of the app calling the API and `HMAC` is a cryptographically secure keyed-hash message authentication code (e.g., HMAC-SHA256) in which `SID` is used as "key" and `caller_app_cert_fingerprint` as "message."

The security of this method is bolstered by the fact that

1. Upon installation, Android verifies that an app has been correctly signed.
2. The operating system can securely identify the caller of a framework API [52].
3. No API is provided to get the value of SID.

For these reasons, as long as the developer's private key remains uncompromised, the privacy of the PDID to an app is maintained.

We implemented this modification as complete transparent replacement of the current API used to get the ANDROID_ID. In this way this defense could be deployed without requiring code changes to existing apps. This will necessarily interfere with advertising libraries, which seek to use the ANDROID_ID to track the usage of multiple apps on the same device. However, as explained in Section 3.2.2, the only identifier that advertisement libraries are supposed to use to track users is the Google Advertisement ID. A possible alternative implementation would be providing the PDID to apps trough a separate API.

It is interesting to note that, concurrently (and independently) to the development of this work, Google changed the behavior of the ANDROID_ID to follow our proposed modification. Although the implementation details differ, the functionality achieved by this change is the same. This modification is available starting from Android version 8 [53].

Implementation. We implemented this defense by modifying the Android *SettingsProvider*, the operating system component responsible to deliver the ANDROID_ID value to the running apps. Our modifications consist of approximately 70 LOC added to the original Android code. The added code introduces an average slowdown of 1.497ms when the API to get the ANDRODID_ID is called. The standard Android API caches this value after the first time an app access it, thus we consider this slowdown as negligible.

3.7 Limitations and Future Work

While we were able to find a surprising number of vulnerable apps, our system is far from perfect. There are a few conceptual ways in which our system might miss vulnerable apps. The most important one is the inability to influence a change in the user's state stored by the app's backend server. For instance, in some games, the dynamic analysis system would need to effectively play the game and, for example, score points or spend virtual currency.

An important source of error in dynamic analysis is the non-determinism inherent in today's operating systems and apps. Some apps explicitly perform random behaviors, which our Invariant Generation step attempts to remove, but it is by no means perfect. For example, if the non-deterministic behaviors are time-dependent or influenced by network delays, they may produce the same result during the Invariant generation, but not during the other phases. Some previous work has been done to try to have fully deterministic replay of actions (see Section 5.3), but the current state-of-the-art does not handle all the source of indeterminism that our system has to deal with.

One other source of future improvement is in the number of identifiers spoofed and transferred by our system. We used a large set of known identifiers for which we could locate Android APIs, but apps could conceivably invent their own identifiers based on collections of obscure system properties, or implement other means of fingerprinting devices. Finding all possible ways this can happen is an open problem.

We would also like to explore the use of network traffic as part of the Invariant generation, to attempt to more precisely determine when a backend is saving and retrieving user state. In particular, a way to assist with the network traffic analysis, as well as other data sources and sinks, is to use a taint-tracking-based analysis system, such as TaintDroid [54]. Unfortunately, we have noticed that many identifiers are sent to the app's

backend, even if they are not directly used for authentication purposes, which represents a significant source of noise for this kind of analysis. This is potentially done to aid in gathering metrics about apps, or to aid in advertising.

Finally, an interesting future work would be to study if the authentication problems we have identified in this work also affect applications running in other mobile operating systems.

3.8 Conclusions

In this work, we explored the real-world vulnerabilities of apps that authenticate their users using device-public information. Some app authors appear to make the assumption that this information is somehow hard to obtain or spoof.

To disprove this, first we developed an "identity-transfer" attack that can be automatically applied to any apps relying on device-public information to authenticate its users. Then, we developed a system, based on dynamic analysis, that infers information about the apps' backend states to locate insecure authentication mechanisms, and perform our attack against them. After analyzing 1,000 popular apps from the Google Play market, we found 41 that were vulnerable to our generic identity-transfer attack, including two major messaging apps used with hundreds of millions of installations.

Finally we proposed and implemented solutions to the identified problems, requiring minimal modifications to the Android operating system and no modifications to the existing apps.

Chapter 4

Hardware-Assisted Authentication

As smartphones become widely used, more and more security-sensitive tasks are performed using these devices. For instance, mobile payment or mobile banking applications have been steadily increasing for the past few years [55]. That is, smartphones are increasingly used to access remote accounts containing valuable and sensitive user information such as purchase histories or health data. Needless to say, the security of smartphones and mobile apps, including authenticity, integrity, and confidentiality, is of paramount importance.

Smartphone technologies bring both new opportunities and threats to security. A smartphone is a very convenient choice to be the "second factor" in two-factor authentications (2FA) because the users do not have to carry additional security tokens. A very common two-factor scheme is to authenticate a user based on both the user's password and proof that the user is in possession of her smartphone, with the latter commonly achieved by sending text messages to the registered smartphone. On the other hand, as more and more sensitive operations that are protected by 2FA are performed using smartphones, the security threat from a stolen/compromised phone or malicious apps running

on the phone significantly increases. In particular, by performing sensitive operations on smartphones, both factors required by 2FA will be available on the smartphone, making it a single point of failure.

In theory, technologies commonly available on modern smartphones can be used to implement 2FA schemes that are secure even in the face of stolen/compromised phones or malicious apps running on the phone. In particular, most smartphones already come with Trusted Execution Environments (TEE) that can be used to generate and store cryptographic keys.[1] Furthermore, the TEE can already be programmed to directly communicate with a fingerprint reader (which is widely available on modern smartphones) so that it will only perform operations using the stored keys when the fingerprint reader detects a registered fingerprint, signaling the user's explicit consent to such operations. Since the TEE is a hardware-enforced isolated execution environment, the keys it stores and the operations performed with those keys cannot be leaked or misused even if the smartphone's operating system (OS) is compromised.[2]

A second factor implemented by combining the TEE and the fingerprint reader is at least as strong as what proposed in the Security Key protocol [58] (and implemented by YubiKey [59]), the current state-of-the-art authentication solution in the desktop world, promoted by Google, as a member of the FIDO Alliance [60]. Under the Security Key protocol, a cryptographic private key is stored on an external hardware device and is used to *sign* authentication tokens provided by the remote service the user wants to authenticate with. This signing operation only happens if the user authorizes it, by pressing a physical button on the external hardware device. In fact, one can argue that

[1] In devices running Android, the TEE is typically enforced by using the ARM TrustZone technology [56].

[2] As an empirical measure, among all the vulnerabilities mentioned in the "Security Bulletins" released by Google about Android security [57] up to August 2017, 33 of them allow an attacker "to execute arbitrary code within the context of the kernel," whereas only 2 allow "to execute arbitrary code in the TrustZone context."

a second factor that combines a smartphone's TEE and its fingerprint reader is going to provide more security than YubiKeys in the scenario where the hardware security token is stolen; in the former, the attacker cannot misuse the hardware token without the owner's fingerprint, while in the latter, anybody in possession of the token can misuse it to bypass 2FA. Additionally, the device's screen (which is not present in standard hardware tokens), could be used to inform users about the operation they are authorizing by touching the sensor.

Motivated by the significant security benefits that the TEE-backed fingerprint sensor can offer, in this work we perform the first comprehensive study on the usage of the fingerprint API in Android. In particular, we first systematically explore the various nuances of this API, and we uncover several aspects, many of which subtle, that can lead to this complex API to be misused. As an example, developers could just check if the user touched the sensor, without binding this operation to the usage of a cryptographic key, contrary to what is suggested by Google's guidelines [61].

We then bring some clarity to the many threat models that should be considered when performing security evaluations concerning the fingerprint API. For example, we explore what are the capabilities of an attacker that can compromise the untrusted operating system, i.e., a "root attacker." At first glance, one may say that a root attacker will trivially defeat any fingerprint API and that the fingerprint API itself is not designed to protect from root attackers. On the contrary, *we argue that many important design choices related to this API are motivated specifically to protect from root attackers.* The most significant example is that current implementations of the fingerprint API work by unlocking a TEE-backed cryptographic keystore: if the threat model were not considering root attackers, apps could simply store cryptographic material in app-private storage (that non-root attackers cannot access), without needing to rely on any TEE support.

We hypothesized that the lack of clearly stated design goals and, as we will see, misleading documentation bring confusion and app developers might misuse this API. To explore this hypothesis, we first developed a static analysis approach to characterize how Android apps use the fingerprint API, whether this API is misused, and how they are resilient to the various threat models. We then use this system to perform the first systematic empirical study of how the current fingerprint API is used in the Android ecosystem. Specifically, we used our tool to analyze 501 apps requiring the fingerprint permission (out of a dataset of 30,459 popular apps). The results are worrisome. For example, the tool identified that 53.69% of the apps, including the widely deployed Google Play Store app, do not make use of the cryptographic keystore unlocked by a successful fingerprint touch: this means that a root attacker can easily completely bypass the fingerprint security mechanism by just programmatically "simulating" the user's touch to, for example, perform in-app purchases.

One explanation for this low percentage could be that not all use case scenarios for the fingerprint API can be protected from root attackers. One example is an app that uses the fingerprint to merely assess user presence: in this case, it is very challenging to find a "role" for the cryptographic material, and it is thus not possible to protect this use case from root attackers. To determine how many apps fall into this category, we then performed manual analysis on a subset of applications flagged as problematic. For example, we manually analyzed a random subset of 20 apps for which our tool identify usages of the fingerprint API flagged as "fully bypassable." To our surprise, 16 of them are apps that use the fingerprint API to authenticate the user against a remote backend, or apps that store secret information: These are *exactly* the use case scenarios that a proper usage of the fingerprint API could easily protect even from powerful attackers such as root attackers. This manual analysis effort, even though admittedly limited,

suggests that the number of apps misusing the fingerprint API is significant. Moreover, our tool also flagged only the 1.80% of the apps in our dataset as using the fingerprint API to *sign* transactions, which is the most secure way to use this API.

In summary, this work makes the following contributions:

- We systematically study the various ways in which the fingerprint API can be used in Android and how attackers with different capabilities can exploit sub-optimal usages of it.
- We develop a static-analysis tool to automatically identify how real-world popular apps use the fingerprint API. We make its code publicly available online [62].
- By using this tool, we perform the first systematic study of the usage of the fingerprint API in Android, and we uncover a significant number of apps potentially misusing the fingerprint API. This improper usage significantly weakens the security guarantees these apps could achieve if using the API correctly.
- We identify shortcomings and weaknesses of the current API and its implementation, and we propose different improvements to it.

4.1 Background

4.1.1 Android Security Mechanisms

The Android operating system is a customized Linux kernel on top of which the Android framework runs. User-installable third-party apps run as user-mode processes and are typically written in Java, even though apps may also include libraries written in native code. These apps interact with the Android framework using system calls or invoking remote procedures in "system services."

Third-party apps run in separate containers with isolated resources (e.g., private files) and a limited set of capabilities. The precise list of capabilities is determined by the "Android permissions" granted to an app. In modern Android versions (starting from Android 6), permissions classified as dangerous need to be specifically approved by the user. Other permissions are instead automatically granted to any app that requires them, but the app still needs to request them in its Manifest file. The `USE_FINGERPRINT` permission, which grants the ability to use the fingerprint reader sensor, is an example of "normal" permission.

This separation between different apps and the different apps' capabilities is enforced by using a combination of standard Linux mechanisms (e.g., Linux groups), SELinux rules, and specific checks in the Android framework. In fact, apps cannot perform any sensitive operation directly, but they have to send a request to a running system service, which verifies whether the app calling it has the permission required to perform the requested operation. Thus, system services (which run as users with higher privilege than normal apps) mediate most of the interactions between apps and the kernel.

Attackers often try to exploit bugs in either the system services or the kernel to gain *root* privileges, using what are typically called "root exploits." Although a significant effort has been made to limit the attack surface exposed by system services and the kernel to normal apps [63], root exploits are still a concrete danger in the Android ecosystem [57]. However, even when an attacker can fully compromise the Linux kernel, achieving persistent kernel-level code execution (by bypassing the Verified Boot mechanism) requires further exploitation of the system [64]. Similarly, achieving code execution within the TrustZone-enforced TEE, which we describe in the next section, requires the exploitation of significantly less common vulnerabilities in the relatively small code base running within the TEE.

4.1.2 TEE and TrustZone

A TEE is an isolated environment designed to execute sensitive code with more isolation/protection than what provided by a standard "feature-rich" operating system. While other instantiations of TEE exist, in this chapter, we will focus on ARM's implementation of the TEE, called TrustZone, which is available on the majority of Android devices.

Under ARM's TrustZone, a "trusted" kernel and a set of Trusted Applications (TAs) run in the "secure" world, isolated by hardware from the Android OS and third-party apps, which, conversely, run in the "non-secure" world. Only code signed by the hardware manufacturer can run in the "secure" world. Also, while third-party apps run in isolation from the TAs, these apps can utilize services provided by the TAs through well-defined APIs. Two services offered by the TAs are relevant to fingerprint-based authentication:

- **keymaster:** It allows to create cryptographic keys, store them inside secure-storage, and use them to encrypt, decrypt, verify, or sign data, coming from the untrusted world. Internally, this service utilizes the secure-storage capability offered by the trusted kernel to securely store encrypted and authenticated data on the device's mass memory.

- **fingerprintd:** It handles the storage of fingerprint data, acquired from the fingerprint reader sensor, and verifies that the finger touching the sensor corresponds to any previously registered fingerprint. It is important to notice that "raw" fingerprint data (i.e., the image of the registered fingerprint) never leaves the TEE and therefore it is not accessible by any untrusted code.

4.1.3 The Fingerprint API in Android

In the discussions that follow, we will focus on apps that access the fingerprint reader (which is commercially named the "Imprint" sensor) through the Java API provided by Google. Unless otherwise specified, we will consider the implementation of this API running in Android version 7 on Google's devices. In particular, for our experiments we used a Google's Nexus 5X.

Also, we will follow Google's [61] and OWASP [65] guidelines and consider that the best way to use the fingerprint reader is in conjunction with some cryptographic operations. In particular, instead of just recognizing the legitimate user has touched the fingerprint sensor, an app should use this fingerprint reading to unlock a cryptographic key protected by the TEE. In other words, by utilizing both the **keymaster** and the **fingerprint** in the TrustZone, this method can guarantee that even an attacker with root privilege cannot misuse the cryptographic key without presenting the right fingerprint. As we will see in Section 4.4, the latter method is significantly stronger.

We will now briefly provide the major steps an app has to perform to interact with the fingerprint sensor and determine whether a legitimate user touched it. For clarity, we will omit unnecessary details of the complex Android cryptographic API, and we suggest interested readers to read the official documentation for a more detailed explanation [66, 67].

Generate a cryptographic key: An app can generate a cryptographic key or a public/private key pair by using the method `initialize` of the class `KeyGenerator` or `KeyPairGenerator`. Developers must specify properties of the generated key (e.g., the algorithm used) by passing a `KeyGenParameterSpec` object to the mentioned `initialize` method.

Among the various aspects a developer can control about a generated key, the most important one in this context is triggered by calling the `setUserAuthenticationRequired` method (passing `true` for its `required` parameter). By calling this method, a developer can ensure that the generated key is usable (i.e., it is "unlocked") only after a legitimate user has touched the fingerprint reader sensor. In case a *pair* of keys is generated, calling this method will only constraint the usage of the private key, leaving the public one freely accessible by the app.

Unlock the key by authenticating the user: By calling the `authenticate` method, an app activates the fingerprint reader sensor. Two parameters of this method are important: the cryptographic key that is unlocked if a legitimate user touches the sensor and a list of callback functions, called after the sensor is touched.

Override the fingerprint callbacks: When a user touches the sensor, specific callback functions are called. In particular, the method `onAuthenticationSucceeded` is called when a legitimate user touches the sensor, whereas other callback functions are called in case of error conditions (e.g., a non-legitimate user touched the sensor).

Use the unlocked key: After the `onAuthenticationSucceeded` method is called, an app should use the now unlocked key. For authentication purposes, Google's guidelines suggest the use of a previously generated private key to *sign* a server-provided authentication token and then send this authentication token to the app's remote backend.

It is worth mentioning two properties of the generated keys. First, the Android framework ensures that only the app generating a key can use it. Second, in modern devices, private keys are stored within the TEE (an app can verify if in a specific device keys are stored within the TEE by calling the `isInsideSecurityHardware` API) and cannot be exported (not even by the app generating them and not even after a legitimate user has touched the fingerprint sensor). In other words, "unlocking" a key does not

allow an app to read its "raw" value, but only to use it to encrypt, decrypt, or sign data. If the key is stored in the TEE, these operations are guaranteed to happen *within* the TEE.

4.1.4 Two-Factor Authentication Schemes

To overcome security and usability limitations of classical username and password authentication, many service providers suggest or mandate the usage of an additional "second factor" during authentication. One common solution is to use a One-Time Passcode (OTP). However, OTPs are still vulnerable to phishing and man-in-the-middle attacks [68, 69] and have serious usability drawbacks, since they require the user to somehow receive the OTP code and insert it into the authentication interface. Furthermore, protocols based on OTPs rely on the confidentiality of the communication channel of the OTP, which is often not guaranteed. For instance, text messages are a common communication channel used to send OTPs to smartphones. However, the insecurity of this channel has been shown in many occasions [70, 71].

Secure authentication schemes using challenge/response offer better security and usability. In particular, the current state-of-the-art is constituted of the Security Keys formalized in the Universal Second Factor (U2F) protocol [72]. This protocol is composed of two phases. During the registration phase, a key pair is generated in an external hardware device. The generated public key is sent to the remote server, whereas the private key remains securely stored within the hardware device. Later, during the authentication phase, the server sends the client a challenge. The client then asks the hardware device to sign this challenge with the stored private key, and the signed response is then sent back to the remote server, which can verify it using the previously obtained public key.

Both during the registration and the generation phases, the user is required to physically touch the hardware device as a Test of User Presence (TUP) to authorize creation and usage of cryptographic keys.

4.2 Threat Model

This section explores the different threat and attacker models considered in this chapter. We first define different "levels of compromise" that an attacker may achieve. Then, we discuss several different threat models, ranging from being just able to install a malicious app on the victim's device to be able to fully compromise the Android Linux (untrusted) operating system. We will also argue why each of these threat models are particularly relevant for any work studying the fingerprint API. We end this section by clarifying which threat models are considered as out of scope.

4.2.1 Levels of Compromise

To ease our exposition, we now define three labels describing three different levels of compromise an attacker can achieve in the different scenarios. We discuss the three levels starting from the least powerful. We note that, of course, an attacker will always attempt to achieve the third and most powerful level of compromise. However, depending on the attacker capabilities and how a given app uses the fingerprint API, this may not always be possible.

Confused Deputy. An attacker might be able to interfere with the usage of the fingerprint API to change the intended effect a user wants to achieve when she touches the fingerprint sensor. For example, consider a user who wants to authorize the transaction "pay \$1,000 to *Friend*" by pressing the fingerprint sensor: an attacker might be able to

change this transaction to "pay $1,000 to *Attacker*." Another example is an attacker that can lure the user to provide the fingerprint by spoofing a completely unrelated scenario, such as the lock screen.

More in general, these examples are instances of a *confused deputy problem*. An attacker can achieve her goal by abusing this problem, but she needs the user to touch the fingerprint sensor *once for each malicious attempt*.

Once For All. In this scenario, the attacker can completely bypass the need for "fingerprint" by just luring the user to provide a fingerprint *once*. That is, after the attacker obtains one fingerprint, the attacker can spoof any subsequent fingerprint request. We note that, in this context, the term "spoofing" does not entail spoofing the "real" physical fingerprint. Instead, with this term, we indicate that an attacker can trick the vulnerable app, and the backend it communicates with, to believe a legitimate fingerprint was provided.

As a representative example, consider an app that, after the user provides a fingerprint, decrypts, using a TEE-backed cryptographic key, an authentication token. If an attacker manages to access this decrypted token, the attacker can now just reuse the token undisturbed for subsequent authentication and authorization attempts, without needing to lure additional fingerprints. Thus, this scenario provides a more practical opportunity for an attacker.

Full Fingerprint Bypass. In this last case, an attacker can completely bypass the need of luring fingerprint touches without requiring a "real" touch, not even once. For example, consider a banking app that requires the user to confirm every monetary transaction by pressing the fingerprint sensor. If an attacker can compromise the app to this last level, the attacker can authorize an unlimited number of transactions, at will, without having

the user touch the sensor. This case provides significant practicality benefits for an attacker. In fact, the attacker does not need to "wait" to hijack a user's touch: as a matter of fact, in this scenario the attack does not need any user interaction at all.

We note that it may not always be possible for a root attacker to indefinitely wait for a user's touch, because, for instance, thanks to the Verified Boot protection mechanism, it may be impossible to persistently compromise a device.

4.2.2 Attacker Capabilities

We consider the following three increasingly powerful attacker capabilities.

Non-Root Attacker. In this threat model, we consider an attacker that is just able to install a malicious application on the victim's device. In this case, we assume that the attacker is unable to subvert the security of the operating system, and therefore the installed malicious app is still constrained by all the limitations imposed by the Android framework. The installed app can, however, request permissions (as any other benign third-party app installed on the device) to obtain specific capabilities, and, in this case, we assume that the user will grant them.

Additionally, the installed app, can show maliciously crafted messages or, more in general, interfere with the device's user interface (UI), to lure a legitimate user to touch the fingerprint reader sensor. These UI attacks greatly vary in terms of complexity and flexibility, and they are well explored by several existing works [73, 74, 1], some of which, such as Cloak & Dagger [75], achieve almost complete compromise of the device. While these attacks are indeed powerful, we note that the fingerprint API might be one of the few aspects that could, at least in principle, prevent full compromise. In fact, even though the Cloak & Dagger attack can simulate arbitrary user input, it cannot "spoof" a physical fingerprint user's touch.

The key conceptual point here is that there is no trusted path from the fingerprint API to the UI. Thus, as previous works have shown, the attacker can exploit an instance of the confused deputy problem. We postpone the discussion on the practicality and implications of these attacks to Section 4.5.2.

Root Attacker. In this threat model, we assume that an attacker can fully compromise the Android operating system, by using, for instance, a "root exploit." Therefore, the attacker can completely bypass apps' restrictions put in place by the Android framework. For example, the attacker can access app-private storage (which is usually protected by the sandboxing mechanism). Moreover, exploiting confused deputy instances via the UI attacks mentioned above becomes much simpler for a root attacker.

Additionally, the attacker can spoof "messages" from the operating system: Specifically, an attacker can freely communicate with the TEE, and thus send arbitrary messages to it. At this point the attacker can programmatically invoke the `onAuthenticationSucceeded` method implemented within the victim app (and thus simulating a user's touch), even if the user has never touched the fingerprint sensor.

We note that, although a root attacker is powerful, she does not get access to everything. In particular, the fingerprint API enforces the following three security properties even on a system in which the untrusted OS is completely compromised:

1. an attacker cannot retrieve "raw" fingerprint data;
2. an attacker cannot retrieve the value of cryptographic key stored into the TEE (i.e., keys are not *exportable*);
3. an attacker cannot use TEE-backed cryptographic keys, unless a legitimate user touches the fingerprint sensor.

However, if the victim app does not properly use such TEE-backed cryptographic keys, the attacker might be able to achieve her goal anyways, as we will explain later.

That being said, we also note that, for some usage scenarios, an app does not have any technical way to secure itself from root attackers. For example, if the app uses fingerprint not to secure a secret or token, but as a local "Test of User Presence" (TUP), there is currently no way a developer could make use of cryptographic algorithms. On the other hand, crypto primitives can be definitively used when implementing remote user-authentication mechanisms. We postpone the discussion about these scenarios to Section 4.5.1.

Finally, for this threat model, we will assume that the device is not in a compromised state when the cryptographic keys ("unlocked" by touching the fingerprint sensor) were first created by the app that the attacker wants to compromise. The creation of cryptographic keys typically happens only during the first usage of an app and, therefore, it may be impossible for an attacker to interfere with their creation if the compromise of a device happens only after this stage of an app's lifecycle.

Root-at-Bootstrap Attacker. In this threat model, we consider an attacker with the same capabilities of the previous one. Additionally, we also assume that the device is in a compromised state even in the moment in which the victim's app generates the cryptographic keys. Therefore, in this case, the attacker can interfere with their creation.

4.2.3 Out-of-Scope Attacker Capabilities

We assume that the TEE is not compromised. In other words, we consider an attacker that can compromise the code running (or the data stored) within the TEE as out of scope. In fact, an attacker able to compromise the TEE can trivially fully compromise the fingerprint functionality, by stealing all the cryptographic keys in the secure storage. Moreover, as previously mentioned, exploits able to gain this capability for an attacker are extremely rare.

We will consider attacks on the physical recognition of the fingerprint as out of scope. These attacks, although possible [76], deal with the physical aspects of the fingerprint acquisition process and with the algorithms used to compare fingerprint data. Conversely, in this chapter, we focus on a higher-level aspect: the operations inside TEE that are triggered by the legitimate user touching the fingerprint sensor, the operating system, and the apps using the fingerprint sensor API. Therefore, we will assume that the fingerprint sensor and the code inside the TEE handling it are always able to understand if the user that is touching the sensor is the legitimate one (i.e., a user who has previously registered her fingerprint as valid using the appropriate operating system interface).

4.3 Fingerprint API usages

In this section, we will explain how the fingerprint API is used by Android apps. In particular, we will classify apps' usages of the fingerprint API based on if and how cryptographic keys (stored inside the TEE) are used to verify that a legitimate user touched the fingerprint sensor. This aspect has profound implications on what attackers can do to subvert the fingerprint checks and how they can achieve their malicious goals. In Section 4.5.1, we will then explain how the verification of the user touching the sensor is used as a part of the authentication schemes implemented by apps and their corresponding backends.

4.3.1 *Weak* Usage

The easiest way to use the fingerprint API is to execute some code after a legitimate user touched the sensor, without using any cryptography. To achieve this, a developer just has to call the `authenticate` method to activate the fingerprint reader sensor and override the `onAuthenticationSucceeded` method to be notified when the user touched it.

From the implementation standpoint, recall that the `authenticate` method takes, as an argument, the cryptographic key that is unlocked when the user touches the sensor (see Section 4.1.3). Thus, an app can set this parameter to `NULL` and, as a side-effect, the fingerprint will not unlock any cryptographic keystore. Of course, an app could also require access to the keystore and it could then discard this object without using it. In other words, a specific fingerprint-protected functionality is not "protected" by cryptographic operations if a cryptographic key is unlocked but never properly used.

4.3.2 *Decryption* Usage

In this case, a cryptographic key is created, stored inside the TEE, and used to decrypt (once the key is "unlocked" by a legitimate user touching the fingerprint sensor) locally stored files. Google's guidelines suggest using the fingerprint API in this way when "securing access to databases or offline files." In practice, we have seen this method often used to decrypt an authentication cookie stored in an encrypted vault within the app's private storage. This authentication cookie, typically valid for multiple sessions, can be used by the app to authenticate with the remote server.

We have found two ways in which this mechanism is implemented. The easiest case is when a *symmetric* key is created and used to encrypt/decrypt the content of the "encrypted vault." The disadvantage of this method is that it requires the user to touch

the sensor (to "unlock" the key) to both *read* something from the vault and to *write* something into it. As a consequence, if, for instance, the remote backend decides to change the value of the authentication cookie stored inside the vault, the user would need to touch the fingerprint sensor to unlock the key.

A more user-friendly way is to use an *asymmetric* key pair. In this case, the public key (which does not need to be "unlocked" before usage), is used to *write* inside the vault, and the private key (which requires the user's touch) is only used to *read* from the vault (e.g., when the stored authentication cookie is needed to authenticate with the app's backend).

Surprisingly, the example officially provided by Google [77] about using the fingerprint API together with a symmetric key does not show how to use cryptography safely. In fact, the provided code generates a symmetric key and, after the user touches the sensor, uses it to encrypt a fixed, hardcoded string. Then, the code just checks whether the encryption operation (performed using the `doFinal` API) threw an exception, an indication that the used key is (still) locked (i.e., it has not been unlocked). While the intent might have been to verify that the user has touched the sensor, this particular example code makes the usage of cryptography pointless because an attacker with "root" privileges can just fake the result of the decryption operation and clear the thrown exception (as we will describe better in Section 4.7.1). In practice, in terms of security, we consider the Google's example on how to use symmetric keys as a case of *Weak* usage of the fingerprint API, rather than a case of *Decryption* usage.

4.3.3 *Sign* Usage

The fingerprint API can also be used to implement challenge/response authentication schemes. This offers significantly more security over a wide range of attackers, but, unfortunately, it is rarely used by developers.

In this case, typically during the app's first usage, a key pair is generated: the public key is sent to the app's remote backend server, whereas the private one is stored within the TEE. When the app needs to authenticate a user to the remote backend, the following steps take place:

1. The remote backend sends a challenge to the app.
2. The app calls the `authenticate` API to "unlock" the previously stored private key.
3. The legitimate user touches the fingerprint reader sensor, and the private key is "unlocked" by the TEE.
4. The `onAuthenticationSucceeded` method (overridden by the app) is called.
5. The app uses the now-unlocked private key to *sign* the challenge from the app's backend.
6. The app sends the signed challenge to the backend.
7. The backend verifies the signature on the challenge, using the public key previously obtained from the client.
8. The backend communicates to the app the result of the verification and considers the user as authenticated.

4.3.4 Sign + Key Attestation Usage

As we discuss in more detail in Section 4.4.3, the "Sign" usage is vulnerable to an attacker that can perform a man-in-the-middle attack at the app bootstrap time, when the initial key exchange takes place. In this attack, the attacker would provide to the backend *her* public key (for which she has the associated private key), and she could then bypass the fingerprint. However, starting from Android 7, a countermeasure to this attack is possible, since Android can provide an "attestation" certificate chain, attesting that a key has been created by a "trusted" TEE. A similar attestation mechanism is present in the Security Keys protocol [58].

To enable it, a developer, when creating a key pair, has to call the `setAttestationChallenge(attestationChallenge)` API with a non-`NULL` value for `attestationChallenge`. Then, the app can retrieve the certificate chain, attesting the generated public key using the `getCertificateChain` API. The app's backend can then verify that the root of this chain is signed by a trustworthy Certificate Authority (typically Google). The certificate, among other pieces of information about properties of the generated keys, contains the `attestationChallenge` previously set, allowing the app's backend to verify that the retrieved key was created as a consequence of a specific request.

4.4 Protocol Weaknesses and Attack Scenarios

We will now highlight the weaknesses of each usage scenarios described in Section 4.3. For each identified weakness, we will also determine which classes of attacker (as defined in Section 4.2) can exploit it. Our findings are summarized in Table 4.1.

4.4.1 *Weak* Usage: Fake TEE response

In the *Weak* usage scenario, fingerprint-based authentication is considered successful as long as the TEE communicates that a legitimate touch happened. This message is delivered by the OS to the client app (by invoking the `onAuthenticationSucceeded` method). In this case, any entity that can control/impersonate the OS to deliver such message can successfully authenticate and authorize any transaction to the server, without having to wait for the user to present the fingerprint even once. In other words, *any "root" attacker can achieve Full Fingerprint Bypass against Weak usage by faking OS messages.* Additionally, a non-root attacker can exploit confused deputy problems by mounting UI attacks. Once again, these attacks are possible because of the lack of trusted UI in Android. We also note that these attacks are possible independently from the specific attacker capabilities and from the specific usage scenario. We refer the reader to Section 4.5.2 for more details.

4.4.2 *Decryption* Usage: Replay Attack

In the *Decryption* usage scenario, the TEE is used to decrypt a value (e.g., an authentication cookie), and the same value is communicated to the client app (and the backend server) for every attempt to authenticate or authorize a transaction. In this scenario, *an attacker only needs to capture this value once to then be able to fully authenticate and authorize any transaction any time in the future, by simply replaying this captured value over and over.*

4.4.3 *Sign* Usage: Man-in-the-Middle Attack

In the *Sign* usage scenario, the TEE is used to protect a private key used in a challenge/response scheme. In this scenario, a root attacker cannot easily compromise the system — in a way, she has similar capabilities as a non-root attacker, and she could thus attempt to exploit confused deputy problems via UI attacks.

However, we note that an attacker can launch a man-in-the-middle attack if she can interfere with the "app bootstrap" process, during the initial key exchange. The attack would work in this way: at bootstrap, instead of sending to the backend server the real key output by the TEE, the attacker can use her own key instead. In this way, the attacker can use the key thus registered to answer any future challenge (because the attacker knows both the public and the private key), thus achieving Full Fingerprint Bypass. Clearly, since this attack requires the attacker to have control over when the key exchange is carried out, it is only possible for Root-at-Bootstrap attackers.

4.4.4 Sign + Key Attestation Usage: Key Proxying

The "Sign + Key Attestation" usage scenario significantly raises the bar for attacks, even for a very powerful attacker such as Root-at-Bootstrap attacker. However, from a conceptual point of view, it is possible to attack this usage scenario as well, by performing a so-called *cuckoo attack* [78]. Specifically, while this mechanism attests that a key has been created by the TEE on a user's device with the goal of preventing an attacker from knowing its private value, it cannot prevent an attacker from "proxying" the app's request for creating a key pair to *her* attacker-controlled device and using the TEE of *her* device. We note that this attack scenario presents serious practicality and scalability issues for the attackers. That being said, we will further discuss this aspect in Section 4.8.3, where we propose improvements on the current implementation of this mechanism.

Table 4.1: Summary of attack possibilities with respect to attacker capabilities and fingerprint API usage.

Fingerprint API Usage / Attacker Capabilities	*Weak*	*Decryption*	*Sign*	*Sign* + *Key Attestation*
Non-Root	C.D. [1]	C.D.	C.D.	C.D.
Root	Full	Once	C.D.	C.D.
Root-at-Bootstrap	Full	Full	Full	C.D.

[1] "C.D." stands for Confused Deputy.

4.5 Discussion

This section discusses aspects related to the fingerprint API that are not strictly related to the API itself or to the specific vulnerable "usage scenarios" described above.

4.5.1 Application Contexts

Typically, the fingerprint API is used as a part of an authentication scheme. In this section, instead of focusing on how apps use the fingerprint sensor in terms of API calls and encryption, we will discuss common functionality apps aim to accomplish when they use the fingerprint sensor.

"Local-Only" Usage. Some apps use the fingerprint API to implement the "screen-lock" functionality. For instance, they prevent access to a list of user-selectable apps, unless the fingerprint sensor is touched by a legitimate user. In this case, the fingerprint sensor just constitutes a local Test of User Presence (TUP).

For these apps, only a *Weak* usage of the fingerprint API is reasonable. In fact, the app does not have any remote backend to authenticate with nor it stores any secret data.

Remote User-Authentication. More interestingly, in many cases, the fingerprint API is used as one part of an authentication scheme. Upon first usage, apps have to provide a single-factor or multi-factor user authentication system, since no cryptographic key is created and stored by the app inside the TEE yet. On subsequent usages, the app (and the corresponding backend) may require the user to touch the fingerprint sensor. Some apps can be configured to require the user to touch the sensor every time the app is opened and it connects to the remote backend. Others ask for this action before performing any sensitive operation, such as a payment.

Typically, when the fingerprint functionality is enabled, the app will allow the use of a fingerprint touch instead of inserting the account's password. While this is convenient in term of usability, it has mixed security consequences. As a security benefit, an attacker achieving "root" cannot steal the account password, since the user is not asked to insert it. However, as we will explain in Section 4.5.2, even a non-root attacker can potentially lure a user to touch the fingerprint sensor and, compared to phishing a password, stealing a fingerprint touch is significantly easier. In fact, touching the fingerprint sensor is a common action, since it is used, for instance, very frequently to unlock the phone. Therefore an attacker can just pretend to be the lock-screen without raising much suspicion. Secondly, a fingerprint touch requires less user's effort and time to be performed and therefore is more likely to happen. Finally, an attacker does not need to ask for a *specific* password, but just to generically touch the sensor.

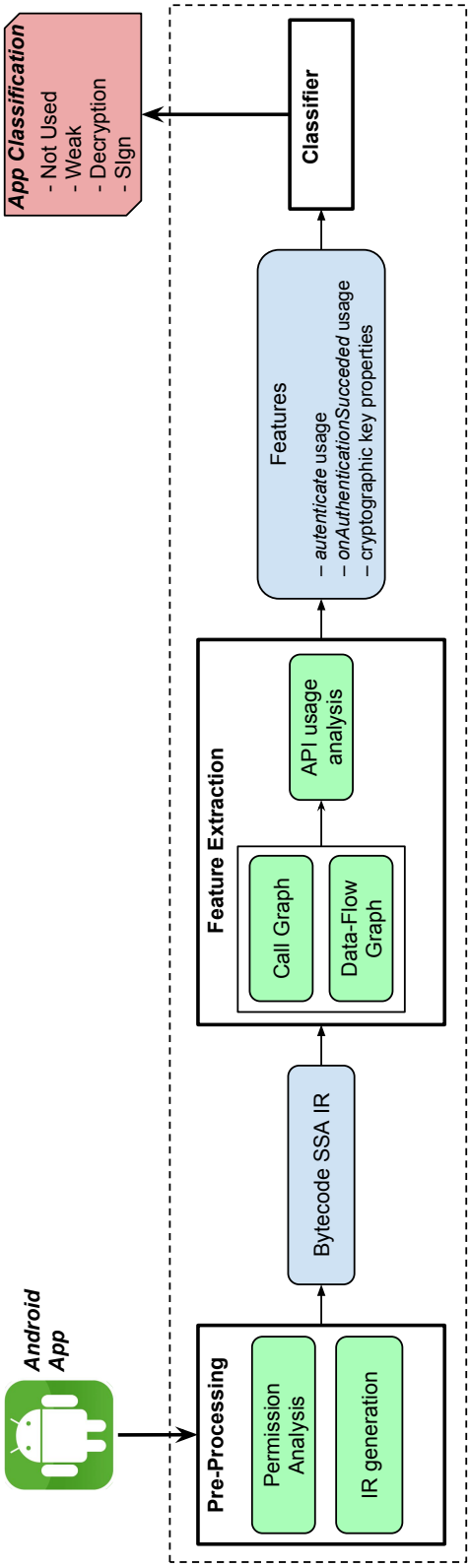


Figure 4.1: Overview of the developed static analysis tool

4.5.2 Practicality and Impact of UI Attacks

As we mentioned earlier, a malicious app can show maliciously crafted messages or, more in general, interfere with the device's user interface to lure a legitimate user to touch the fingerprint reader sensor. In particular, we mentioned how several existing works [73, 74, 1] show the possibility to perform UI attacks, and that a very recent work, dubbed Cloak & Dagger [75], can achieve almost complete compromise of the device. In particular, this last work showed that apps installed from the Play Store are automatically granted the `SYSTEM_ALERT_WINDOW` permission (which allows to create overlays windows on top of any other) and that it is possible to lure the user to unknowingly grant accessibility permissions to a malicious app through "clickjacking."

These attacks are powerful, especially because they can be performed by any unprivileged app (what we refer to as "non-root attacker"). However, we note that the fingerprint API might be one of the few aspects that could, at least in principle, prevent full compromise: a physical fingerprint "touch" cannot be spoof via UI-only attacks.

That being said, there are many attacks that one could perform. These attacks are all instances of a confused deputy problem, and they are all possible due to one key observation: no "Secure UI" is currently used by the fingerprint API, and the user does not have any mechanism to establish with which app she is interacting with. As a very practical example of these attacks, Zhang et al. [79] show how an attacker can create a fake "screen lock" to lure the user to provide her fingerprint: the fingerprint, under the hood, is actually "passed" to a security sensitive app in the background.

More in general, the lack of "secure UI" allows an attacker (independently from the fingerprint usage scenarios described in Section 4.3) to lure the user to present her fingerprint believing she is authenticating with app A or authorizing transaction X, while the fingerprint is actually used to unlock keys for a different app B or to authorize transaction Y.

These attacks are affected by practicality aspects. First of all, an attacker needs to solve two issues:

1. Put the victim app in a state in which, once the fingerprint sensor is touched, an unwanted malicious action happens.

2. Lure a legitimate user to touch the sensor.

Second, the attacker needs to steal a fingerprint touch every single time she wants to perform the attack. However, this last challenge can be easily addressed: since the fingerprint is often used to perform "screen unlock" and since the "screen unlock" action is an action that a user is used to perform tens of times every day, it is straightforward for an app to create a situation for which the user would provide a fingerprint.

From a technical standpoint, an attacker can exploit this by simulating that a device got automatically locked (which, by default, happens after a few seconds of non-usage). To achieve this, the attacker can show a fullscreen, black overlay on top of any existing Activity.[3] Moreover, by requiring the permission `WRITE_SETTINGS`, the attacker can also minimize the background light of the screen. At this point, the attacker can prevent the device from automatically locking itself (by using the `WakeLock` API, requiring the automatically-granted `WAKE_LOCK` permission). In this scenario, a user will likely assume that the device got automatically locked and try to unlock it by touching the fingerprint sensor.

[3] An Activity is the standard "unit of interaction" in Android and loosely corresponds to a window in a desktop environment.

As an attempt to defeat these UI attacks, a countermeasure is currently implemented by the Android framework. Specifically, an app can only request the usage of the fingerprint sensor if it is displayed in the foreground. Unfortunately, in evaluating if an app is in the foreground, the Android framework only evaluates its position in the Activity stack. Since the Android framework does not deem screen overlays as part of the Activity stack, an Activity will still be considered as in foreground, even when maliciously covered by an overlay.

4.6 Automatic Analysis Tool

We have developed a tool to automatically analyze how an app uses the fingerprint API. The tool takes an Android app as input and classifies its usage of the fingerprint API into *Weak*, *Decryption*, and *Sign* usage, as defined in Section 4.3. We use the tool above to perform the first systematic study on how Android applications use the fingerprint sensor, pinpointing cases in which this API is incorrectly used. We believe app developers and app market operators can also use this tool to automatically understand if there is any issue in how an app uses the fingerprint API. Figure 4.1 provides an overview of the developed tool.

4.6.1 Challenges and Design Choices

Our tool performs static analysis on an app's bytecode. We choose static analysis on bytecode to be able to perform our analysis without needing source code (which is typically unavailable both to security researchers and market operators). Moreover, many apps using the fingerprint API belong to the "finance" category. This makes very difficult to automatically perform dynamic analysis on these apps, since we do not have the required financial account information needed to get past the login stage. Even

approaches able to automatically register accounts while performing dynamic analysis, such as AppsPlayground [41], cannot solve this problem by automatically creating bank (or other financially related) accounts. This aspect also significantly complicates our manual investigation of the results and our attempts to dynamically execute a given app.

One of the main challenges when analyzing recent real-world Android apps is the amount of code these applications include (on average, the apps we have analyzed have about 51,000 methods). This is often because apps include big libraries, which, even if only marginally used, substantially increase the amount of code a static-analysis tool may end up analyzing. Empirically, recent research [80] has shown that even relatively easy data-flow analysis, such as flow-insensitive taint analysis, often ends up using unpractical amounts of resources and time, when applied to an entire app. However, for the analysis we are interested in, we only need to precisely characterize the usage of very specific API methods. For these reasons, we adopted a more localized approach, which constructs call graph and data-flow graphs starting from the APIs of interest, limited to the specific parameters we are interested in.

4.6.2 Pre-processing

The first step of our analysis is to determine which apps potentially use the fingerprint API. Since, to use the fingerprint hardware, an app has to require the `USE_FINGERPRINT` permission, our tool first checks whether a given app requires this permission by reading its manifest file. Apps not requesting this permission cannot use the fingerprint API.

After this step, we use the Java static analysis framework SOOT [81] to obtain an intermediate representation of the app's bytecode. To simplify further data-flow analysis, we choose the *Shimple* intermediate representation, which is in single static assignment (SSA) form.

4.6.3 Call Graph Construction & Data Flow Analysis

Our analysis is based on two static analysis primitives: call graph generation and data-flow graph analysis. The call graph represents method invocations among different methods in the analyzed app. In building the call graph, we perform intra-procedural type-inference [82] to determine the possible dynamic types of the object on which a method is called. If this analysis fails, we over-approximate the possible dynamic types as all the subclasses of its static type (including the static type itself).

Our call graph also considers some implicit control flow transitions introduced by the Android framework [83]. In particular, when the `onAuthenticationSucceeded` callback is invoked by the Android framework, typically developers call the `postDelayed` method, by passing, as parameter, an instance of a specific inner-class, implementing the `Runnable` interface. On this inner-class, the method `run` will be later called and executed in a different thread. This is a common behavior in Android, since code dealing with UI elements has to run in a different thread than code dealing with network operations, to ensure app's responsiveness.

Our tool handles these cases by identifying the possible dynamic types of the instance passed to the `postDelayed` method. Then, it adds edges in the call graph between the `postDelayed` method and the implementations of the `run` methods that can be possibly called, according to the identified types (typically, just one).

To perform data-flow analysis, starting from a variable of interest V (e.g., a specific parameter of an API call), we recursively follow the def-use chain to obtain an *inter-procedural* backward slice. Moreover, when a field access is encountered, we continue the analysis starting from all the instructions accessing it. As an output of this analysis, we obtain a slice of instructions (encoded as a tree) in which each instruction uses variables that may influence the value of V.

Table 4.2: Overview of the collected features

`authenticate`	*Null*/*NonNull*
`onAuthenticationSucceeded`	*NoCrypto*/*Constant*/*Decrypt*/*Signature*
Key Properties	*DecryptionKey*/*SigningKey* *UnlockedKey*/*LockedKey*

4.6.4 Feature Extraction

At a high-level, our analysis extracts three kinds of features:

1. how the `authenticate` API is used;
2. which code is triggered when the `onAuthenticationSucceeded` callback is called;
3. the parameters used to create cryptographic keys.

Table 4.2 enumerates the features we extract to characterize these three aspects.

***authenticate* API Usage.** For the `authenticate` API, for each occurrence of a call to this method, our analysis generates a backward slice, starting from the parameter named `crypto`. This parameter is used to specify the cryptographic key that is "unlocked" whenever a legitimate user touches the fingerprint sensor. Then, by analyzing the generated slice, we check if the value of this parameter is `NULL`. In this case, it means that the `authenticate` API will activate the fingerprint sensor, but no key will be unlocked when the user touches it. We mark this case as *Null*, otherwise we mark it as *NonNull*.

***onAuthenticationSucceeded* Callback Usage.** We analyze the code that is executed when the `onAuthenticationSucceeded` callback is invoked, to determine if and how cryptographic operations happen after the user touched the fingerprint sensor. Starting from each occurrence of a method overriding `onAuthenticationSucceeded`, we start a forward exploration of the call graph, looking for calls to specific cryptographic methods.

Specifically, if we encounter a call to the methods `sign` or `update` of the class `Signature`, we mark this usage of `onAuthenticationSucceeded` as *Signature*, whereas if we encounter a call to the methods `doFinal` or `update` of the class `Cipher`, we mark it as *Decrypt*.

As a special case, if after the `onAuthenticationSucceeded` callback a decryption operation is detected, but it is performed on a fixed, hardcoded string (as explained in Section 4.3.2), we mark this case as *Constant* (instead of *Decrypt*). To determine this, we generate a backward slice starting from the parameter specifying the decrypted content, and we analyze it to determine if it results in a constant string.

In case we do not encounter any of the aforementioned cryptographic methods we mark the usage of the `onAuthenticationSucceeded` callback as *NoCrypto*, since it shows that no cryptographic operation is performed as a consequence of the user touching the fingerprint sensor.

Cryptographic Key Properties. To determine the type of the used cryptographic keys, we generate a backward slice starting from the `purpose` parameter of the `KeyGenParameterSpec.Builder` constructor. In case we determine it to have the value `PURPOSE_SIGN` we mark the key as a *SigningKey* otherwise we mark it as a *DecryptionKey*.

We also verify if the `setUserAuthenticationRequired` method is invoked (by passing `true` for its `required` parameter). If this is the case, we mark the key as *Locked*, otherwise, we mark it as *Unlocked*.

Other Features. To integrate the information collected by the features just described, we also check if an app is using the `getCertificateChain` and `setAttestationChallenge` APIs. While we do not use this information to classify how an app uses the fingerprint API, we will use this information to study if apps use key attestation (see Section 4.4.4 and Section 4.7.7).

4.6.5 App Classification

After collecting the aforementioned features, we use them to classify how the analyzed app uses the fingerprint API. The rationale behind this classification rules is first to identify cases in which the fingerprint API is not used (e.g., no fingerprint-related API is called) or used in a *Weak* way (e.g., no cryptographic operation is performed). Then, we analyze the properties of the used cryptographic keys and the cryptographic methods called to determine whether to classify the app as *Decryption* or *Sign*.

First of all, we note that for some of the analyzed apps that request the `USE_FINGERPRINT` permission, we cannot identify any usage of the `authenticate` API or the `onAuthenticationSucceeded` callback. We classify these apps, together with those not requesting the `USE_FINGERPRINT` permission, as "Not Used."

Then, we classify an app as *Weak* if any of the following conditions are met:

1. We do not detect any key generation (i.e., the `KeyGenParameterSpec.Builder` API is never used).

2. All the usages of the `authenticate` API are marked as *Null*. This corresponds to the case in which no cryptographic key is unlocked as a consequence of the user touching the fingerprint sensor.

3. All the usages of the `onAuthenticationSucceeded` callback are marked as *NoCrypto* or *Constant*. This corresponds to the case in which no cryptographic operation is performed after the user touched the sensor (or the only cryptographic operation happening is performed on a constant value).

4. An *Unlocked* key is used. In fact, in this case, the used key is not locked, and any root attacker can immediately use it, without having the user touching the fingerprint sensor.

Table 4.3: Static analysis tool results summary

Total Apps	Analysis Errors	Not Used
501	5 (1.00%)	72 (14.37%)

Category	*Weak*	*Decryption*	*Sign*
Detected apps	269 (53.69%)	146 (29.14%)	9 (1.80%)
Misclassifications	0/20	1/10	1/9

At this point, we know that some proper cryptographic operation happens after the user touches the fingerprint sensor. To determine whether the app uses the fingerprint API in a *Decryption* or in a *Sign* way, we use the following rule. We classify an app as *Sign* if any key marked as *SigningKey* is generated and any usage of the `onAuthenticationSucceeded` callback is marked as *Signature*. Otherwise, we classify the app as *Decryption*.

4.7 Automatic Analysis Results

4.7.1 Evaluation Methodology

To determine the correctness of the classification of our tool, we employed the following two-step methodology:

Driving the App to Ask for Fingerprint

In the first step of our evaluation, we manually drive the analyzed app to the point where it starts communicating with the TEE for fingerprint-based authentication.

One significant challenge in this step is that most of the considered apps require specific accounts to go beyond the initial login interface, and it is impractical to create accounts for many such apps. This is because many of the apps we analyzed are mobile-

banking apps, for which it is not possible having an account without also being customers of the connected bank. In other cases, the app's backend requires financial information such as Social Security Numbers or debit card numbers to create an account, which further hindered our ability to interact with these apps.

Verify the Existence of Expected Weaknesses

Once we drive the analyzed app to start interacting with the TEE, we verify our tool's classification for this app by simulating a root attacker and see if the fingerprint-based authentication is vulnerable to weaknesses of the corresponding class as predicted in Section 4.4. For simulating a root attacker, we used the Xposed Framework [37], a tool which allows us to easily modify apps' and framework's Java code at runtime.

In particular, if our tool classifies the app as using the *Weak* usage, our simulated attack modifies the behavior of the `authenticate` API to directly call the `onAuthenticationSucceeded` callback. Furthermore, we deal with the case in which the victim app invokes any cryptographic operation using a key stored inside the TEE. In this case, the app would raise an exception, since this key has not been "unlocked." This scenario may occur in the case in which the result of the decryption is not used (and therefore we classify the app as *Weak*), but still, a TEE-protected key is used to decrypt a hardcoded string, as it happens, for instance, in the Google's sample code [77]. We deal with this case, by masking the generated "User Not Authenticated" exception.

For apps classified as using the fingerprint API in a *Decryption* way, we first record the outputs of decryption operations using TEE-protected keys (simulating a Root attacker). Then, we modify the `authenticate` API as explained before and, additionally, we replay the collected decryption outputs when necessary.

4.7.2 Dataset

We collected all the free apps classified as "Top" (i.e., most popular) in each category of the Google Play Store. These apps were downloaded in February 2017. Additionally, we added apps preinstalled on a Nexus 5X device running Android 7. In total, we created a dataset of 30,459 apps. Among these apps, 501 (1.64%) declare the `USE_FINGERPRINT` permission and, therefore, can potentially use the fingerprint API. In the rest of this section, we will focus on this subset of 501 apps.

4.7.3 Apps Classification

Table 4.3 summarizes the outputs of our tool. We ran our tool in a private cloud, and for the analysis of each app we provided 4 virtual-cores, 16 GB of RAM and 1 hour time limit. For the 501 apps, our tool needed on average 354 seconds ($\sigma = 363$) of computation and used 6.13 GB ($\sigma = 1.07$) of RAM per app. In 5 cases (1.00%), our analysis did not finished due to bugs in the SOOT framework or analysis timeout.

For 72 (14.37%) apps, although they ask for the `USE_FINGERPRINT` permission, our tool did not detect any usage of the fingerprint API. This result is not particularly surprising since previous research has shown that apps tend to require more permissions than they use [84]. To further verify this finding, we manually analyzed a random sample of 10 of these apps. We both manually run them and perform tool-assisted reverse engineering. For 7 of them, we could confirm that they do not use the fingerprint API, whereas for the other 3 our tool was unable to detect its usage because these apps use native code components to activate the fingerprint reader sensor, which our tool is unable to analyze.

For apps classified as *Weak* we took a random sample of 20 apps among those in which we were able to dynamically reach the fingerprint interface. Our dynamic analysis confirmed that they were all correctly classified (i.e., our simulated attack in Section 4.7.1

is successful). Among these 20 apps, 16 access a remote account or store secret data, therefore a *Weak* usage of the fingerprint API is not appropriate (as explained in Section 4.5.1).

For apps classified as *Decryption* we took a random sample of 10 apps and we confirmed that 9 were correctly classified (using, again, the simulated attack explained in Section 4.7.1), whereas 1 was classified as *Decryption* while in reality is *Weak*.

Finally, about the 9 apps classified as *Sign*, we were able to dynamically reach the fingerprint interface in one app and dynamic analysis confirmed the classification of this app as correct. This app, called "True Key," requires to *sign* an authentication token during login and performs this operation with a TEE-protected private key, "unlocked" only when the user touches the fingerprint reader sensor. To have a better evaluation, we also extensively reverse engineer the other 8 samples classified as *Sign*. Our manual analysis revealed that 7 of them have been classified correctly, whereas 1 has been classified as *Sign* while being *Decryption*.

In summary, we manually analyzed (either by reproducing our attacks as explained in Section 4.7.1 or by reverse engineering) 39 apps and we found that all the apps except 2 were classified correctly. In one case the misclassification is due to overapproximations in the call graph. In the other, the app "signs" some data, but this data is constant, since it is provided by the backend when the user logins the first time. For this reason, the app falls into the *Decryption* category. In fact, an attacker can trivially replay the result of this signing operation after it happened once. However, our tool was unable to detect this scenario and, therefore, it classified the app as *Sign*.

Overall, results show how our tool is reasonably accurate in determining how an app uses the fingerprint API. Moreover, the few misclassifications "overestimate" the security of an app (classifying it as using the fingerprint API in a stronger way than in reality). Therefore, we believe that our results, showing a low usage of the fingerprint API in the

Sign way and a high usage in the *Weak* way, are particularly worrisome and confirm our intuition that apps generally do not use appropriately the fingerprint API. In the next sections, we will provide concrete examples of these inappropriate usages.

4.7.4 Case Study: Unlocking "Unlocked" Keys

As explained in Section 4.1.3, a key is stored inside the TEE and "unlocked" by a fingerprint touch only if the `setUserAuthenticationRequired` method is invoked (by passing `true` for its `required` parameter) when the key is generated. On the contrary, without calling this method, a generated key is always "unlocked," regardless of the usage of the fingerprint API.

Surprisingly, we found this aspect as a source of implementation errors. In particular, we looked for apps implementing proper cryptographic operations as a consequence of the user touching the fingerprint sensor (i.e., calling the `authenticate` API to "unlock" a key used to decrypt or sign some data), but not calling properly the `setUserAuthenticationRequired` method. This indicates that the developers wanted to have a key "unlocked" when the legitimate user touches the fingerprint sensor, but forgot to "lock" the key in the first place.

To identify these apps, we checked for apps that

1. are classified as *Weak* by our tool;
2. do not call the `setUserAuthenticationRequired` method (or they call it specifying `false` as its parameter);
3. if they had called the `setUserAuthenticationRequired` method properly they would have been classified as *Decryption* or *Sign*.

Our tool identified 15 apps in this scenario and we were able to fully dynamically interact with 4 of them, verifying their improper usage of the fingerprint API.

As an example, one of these applications allows a user to purchase items in an online marketplace and requires the user to touch the fingerprint sensor during the checkout procedure. The user's password is stored encrypted by a supposedly TEE-secured key, as is common when the fingerprint API is used in a *Decryption* way. During the checkout, when the user touches the fingerprint sensor, this key is used to decrypt the user's password. However, we verified that the decryption key is not really "locked" since the `setUserAuthenticationRequired` method is not called. Therefore, from a cryptographic perspective, the use of the fingerprint API is useless. As a consequence, a root attacker can easily bypass its usage.

4.7.5 Case Study: Google Play Store

Among the apps our tool classified as *Weak*, one is the "Google Play Store" app. This app is present on every Google-branded phone, and it handles the purchase of apps, media, and in-app purchases and can be setup to "protect" these purchases by a fingerprint touch. In this case, the user would be required to touch the sensor before every purchase. Since this app can directly spend user's money and interacts with a remote server, the most appropriate usage of the fingerprint API would be *Sign*, as also stated and exemplified in the guidelines from Google itself.

However, our tool classified the Google Play Store app as using the fingerprint API in a *Weak* way and our evaluation (as described in Section 4.7.1) confirmed this result. In fact, this app calls the `authenticate` API with a `NULL` value for its `crypto` parameter, and, therefore, no key is "unlocked" and no *sign* operation certifies that the purchase happened as a consequence of the user touching the fingerprint reader sensor.

On July 2017, we contacted the Android's security team. The team promptly replied and forwarded our report to the Google Play's team, which is now aware of the issue and investigating it.

4.7.6 Case Study: Square Cash

Among the apps our tool classified as *Decryption*, one is the "Square Cash" app. This app is a personal payment app, which allows users to transfer money to and from connected debit cards and bank accounts.

The app can be configured to require the user to touch the fingerprint sensor before any transaction. The most appropriate usage of the fingerprint API in this case would be to use it to *sign* these transactions. However, Whorlwind, the open source library that Square (and other apps in our dataset) uses to implement the fingerprint functionality, implements a weaker scheme. In particular, this library is used to decrypt a locally stored authentication token. For this reason, by simulating an attacker with Root capabilities, we were able to reuse the same decrypted token to perform different payments.

We contacted the developers of the Whorlwind library in August 2017, detailing our findings and why we think that a *Sign* usage of the fingerprint API is more appropriate in this case.

4.7.7 Case Study: Key Attestation

We mentioned in Section 4.4.4 that, starting from Android 7, a new mechanism has been implemented to allow developers to "attest" public keys, ensuring they have been generated from "trusted" TEEs. According to the API, a properly ver-

ified certificate chain, "rooted at a trustworthy CA key," is only provided if the `setAttestationChallenge` API, with a non-NULL value for `attestationChallenge`, is called.

Conceptually, apps using both the fingerprint API in a *Sign* way and key attestation should be categorized in a different group in Table 4.3. However, in our dataset, our tool found no app calling this API. This indicates that every app in our dataset is vulnerable to a Root-at-Bootstrap attacker, who can interfere with the initial key exchange process between the app and its remote backend.

4.8 Fingerprint API Improvements

We will now propose some changes to the current fingerprint API, which would significantly improve its security. In this section, we will assume that apps use the fingerprint API in a *Sign* way, which, as previously shown in Section 4.4, it is the right way to provide stronger security. However, even with proper usage, this API currently has some shortcomings, which we will address here.

4.8.1 Trusted-UI

The biggest limitations of the current API and its implementation are:

1. Users have no *trusted* way to understand what they are *signing* by touching the fingerprint sensor.

2. A malicious application (with or without "root" privileges) can interfere with what is shown to the user when asked to touch the sensor.

To solve both issues, we propose a mechanism in which the TEE can directly show to the user the content of a *sign* operation performed by a fingerprint-unlocked key. This mechanism is based on the known idea of having a *trusted* video path directly between the TEE and the device's screen. TEE-enforced video paths are already implemented in some Android devices (for DRM purposes) [85] and academia explored its use for authentication purposes [86]. However, differently from previous solutions, what we propose is also based on a *trusted* input which is the fingerprint reader sensor, able to directly communicate with the TEE.

We propose to change the current `authenticate` method to also take as an input a `message` string parameter, for instance "Do you want to authorize a payment of $1,000 to *Friend*?" This message would be shown on a TEE-enforced Secure UI dialog window, alongside with a standardized graphic UI asking the user to touch the fingerprint sensor. *Untrusted* code, outside the TEE, cannot interfere with the visualization of this window, due to the usage of a secure video path. Specifically, *untrusted* code cannot read the content of this dialog window nor modify it.

When the sensor is touched by a legitimate user, a signature of this string (generated using the private key "unlocked," specified when the `authenticate` method is called) is available using a method called `getSignedMessage`. The remote backend can then verify that this message has been signed correctly and, therefore, be sure of what the user has authorized by touching the sensor. In other words, the remote backend can verify the "user intention," which is signed by the TEE.

The security of this system is guaranteed by the fact that both the code for handling the *sign* operation and the code for visualizing the message are within the TEE. Therefore, an attacker, even having root privileges, cannot decouple what is being shown to the user with what is being signed by the fingerprint-unlocked key. An attacker can still interfere with the communication between the backend, the app, and the TEE. However, this will

be detectable by the user. In fact, suppose that the attacker changes the request the app sends to the backend from "Pay *Friend* $1,000" to "Pay *Attacker* $1,000." As a consequence the backend will send the following message to be signed by the TEE: "Do you want to authorize a payment of $1,000 to *Attacker*?". In this case, the user will be able to notice that the message does not correspond to her intention.

Another issue is how to prevent an attacker from showing a malicious dialog window that resembles the window shown by the TEE when asking the user to touch the fingerprint sensor. Without requiring extra hardware (e.g., an LED would be turned on when "secure output" is displayed), we can exploit the fingerprint sensor itself to mitigate this attack. Since the fingerprint sensor can communicate directly and exclusively with the TEE, we propose that the TEE shows a hard-to-spoof visual clue (e.g., a loading bar) while the user touches the sensor.

Attackers would be unable to show this bar at the right time, since, outside the TEE, it is unknown when the user touched the sensor. Therefore, the absence (or the improper behavior) of this visual element would indicate to the user that the shown dialog window is not legitimate. Another possible solution, although less practical since it requires a setup phase, would be to use a secret (i.e., only know by the user and the TEE) personalized security indicator. This mechanism has been shown as an effective defensive mechanism in the Android ecosystem [1].

It is important to notice, however, that even without this defense, an attacker would not be able to lure users to *sign* a malicious transaction, but only to pretend that a transaction happened.

4.8.2 Other UI Changes

While a solution based on hardware-enforced secure-UI is the best way to address current API shortcomings, we understand that its adoption and deployment may be problematic because it requires non-trivial modifications to the code running inside the TEE and the coordination between this code, the Android operating system, and the display hardware. Therefore, we also propose easier-to-implement modifications to the current Android user-level framework. While attackers having "root" privileges can trivially bypass these mechanisms, they are still effective against a non-root attacker.

In particular, Android should automatically dismiss overlay windows on top of interfaces asking the user to touch the fingerprint sensor. A similar solution is already applied in the latest Android versions to protect "security sensitive" interfaces, such as the one used to grant/remove apps' permissions. In addition, the name (and the icon) of the app asking the user's touch should be clearly shown. To implement both solutions, a standard interface, which apps cannot modify except showing some text on it, should be shown when the `authenticate` API is called. In the current implementation, custom interfaces are possible, but uncommon. In fact, most of the apps show very similar interfaces (Android guidelines precisely define how this dialog should appear [87]), thus they will not need to significantly change their UI.

4.8.3 Better Attestation Mechanisms

As we previously mentioned, a key attestation mechanism has been implemented, starting from Android 7. However, in its current implementation state, this mechanism has several weaknesses.

First of all, the API defines two possible "levels" for the attestation "software" and "hardware," where only the latter guarantees that a key has been generated by the device's TEE. The level of attestation can be retrieved by parsing the attestation certificate associated with a generated public key. However, in the devices we have tried (Nexus 5X and Pixel XL, running Android 7), the generated keys are always "software" attested.

More fundamentally, while analyzing the generated certificates, we did not find any indication of the specific instance of the device generating a key. As also pointed out by the paper presenting the Security Key protocol [58], there is a trade-off between user's privacy and security of the protocol. Having a system that can identify the specific device generating a key would allow remote backends to detect suspicious situations in which the key associated with a specific user changes. Moreover, it would hinder the ability of an attacker to "proxy" key creation to an attacker-controlled TEE, since too many keys (used by many different users) generated by the same device would be easily detected as suspicious. However, this would violate user's privacy, allowing unique user's identification among different apps. Therefore, we recommend, as in the Security Key protocol, the implementation of a batch attestation scheme, in which a set of devices, using the same hardware (and potentially affected by the same security issues), shares the same attestation key.

Finally, we note that the current documentation about how to verify key attestation certificates is insufficient and the only official sample code [88] does not cover all the possible cases that need to be handled while parsing this type of certificates.

4.9 Limitations and Future Work

This work focuses on the most common fingerprint API in Android, used by Google's devices. However, Samsung's and Huawei's devices offer their custom fingerprint hardware and a different API. Moreover, outside of the Android ecosystem, similar systems are offered on Apple's devices [89, 90]. Studying similarities and differences among these APIs and how apps that want to be compatible with multiple devices handle this fragmentation is the main future direction of this work.

Our static analysis is based on call graph generation and data-flow graph analysis. This approach has been proved effective by previous research [91] in determining how specific APIs are used in Android. However, this approach is unable to analyze reflective code, dynamically loaded, or native components. Regarding the first two aspects, we do not expect them to be a significant source of imprecision when analyzing non-malicious code and we did not find any sample misclassified because of these reasons. We consider the analysis of components written in native code outside the scope of this chapter. Empirically, we found that the usage of native code prevented us from analyzing three apps (among those manually verified), as explained in Section 4.7.3.

More fundamentally, the implemented static analysis can indicate the way in which an app uses *locally* the fingerprint API, but it cannot fully evaluate how this aspect affects the overall authentication mechanism implemented by the app and its backend. This analysis usually requires probing the remote backend (when, as it is typically, the backend's code is not available) to determine if it properly checks user's authentication. Merging our tool with more general remote protocol analyzers (as, for instance, the one proposed by Zuo et al. [92]) represents another interesting future direction.

4.10 Conclusions

This work provides the first systematic study on the usage of the fingerprint API in Android. We show that its usage is not well understood and often misused by apps' developers. In particular, our study shows that several apps, including popular ones such as Google Play Store and Square Cash, do not use this API in the most secure way. We believe that the fingerprint API could significantly improve the security and the usability of existing authentication and authorizations schemes, especially because the hardware it needs is commonly available in modern mobile devices. We hope this work will highlight current weaknesses and push Google to provide better documentation and to address the remaining problematic issues.

Chapter 5

Related Work

5.1 Attacks to Mobile User-Interfaces

As mentioned in the introduction of Chapter 2, previous papers have already shed some light on the problem of GUI confusion attacks in Android. In particular, [6] describes tapjacking attacks in general, whereas [7] focuses on tapjacking attacks against WebViews (graphical elements used in Android to display Web content). Felt et al. [8] focus on phishing attacks on mobile devices deriving from control transfers (comparable to the "App Switching" attacks we described), whereas Chen et al. [9] describe a technique to infer the UI state from an unprivileged app and present attack examples. Our work generalizes these previously-discovered techniques by systematizing existing exploits and introducing additional attack vectors. We also confirmed the effectiveness of these attacks through a user study. More importantly, we additionally proposed two general defense mechanisms and evaluated their effectiveness.

Fernandes et al. present a GUI defense focusing on keyboard input in [12]: the "AuthAuth" system augments the system keyboard by presenting a user-defined image and the app name and icon. Our proposed defense system uses the same "UI-user shared secret" mechanism: in both cases, users must first choose an image that will be known only by the OS and the user, making it unspoofable for an attacking app.

However, our works significantly differ in how this mechanism is used and what is presented to the user. For instance, as we have shown before (e.g., see Figure 2.4), app names and icons are not valid or reliable roots of trust, as they are easy to spoof. Apps with similar-looking names and icons are commonly present in Android markets, and fake apps with the same name and icon can be side-loaded on the device. Our work, instead, establishes a root of trust to the author of the app and extends the covered attack surface by considering more attack scenarios and methods. In particular, we opted to secure all the user interactions instead of focusing only on the keyboard, because users interact with apps in a variety of ways. For instance, some payment apps (e.g., Google Wallet) use custom PIN-entry forms, while others get sensitive input such as health-related information through multiple-choice buttons or other touch-friendly methods.

Roesner et al. [93] studied the problem of embedded user interfaces in Android and its security implications. Specifically, they focus on the common practice of embedding in an app's graphical elements, created by included libraries. The problem they solve is related and complementary to the one we focus on. Specifically, they focus on how users interact with different elements within the same app, whereas we focus on how users interact with different apps.

Felt et al. performed a usability study to evaluate how users understand permission information shown during the installation process of an app [94]. They showed that current permission warnings are not helpful for most users and presented recommendations for improving user attention. Possible modifications to how permissions are displayed to

users and enforced have also been studied in Aurasium [95]. Our work has in common with these the fact that it proposes a set of modifications to give users more information on the current status of the system, although we address a different threat.

Many studies investigated how to show security-related information and error messages in browsers, both from a general prospective [96, 97, 98] and specifically for HTTPS [99, 100, 101, 102, 103]. Akhawe et al. [103] showed that proper HTTPS security warning messages are effective in preventing users from interacting with malicious websites. The knowledge presented by these works has been used as a baseline for our proposed defense mechanism. It should be noted, however, that other studies have shown that indicators are not always effective. In fact, over the years, the situation has significantly improved in browsers: compare, for instance, the almost-hidden yellow lock on the status bar of Internet Explorer 6 from [102] with Figure 2.1. We believe that our solution may also have benefited from the EV-style presentation of a name in addition to a lock and the consequent increase in screen area. In general, effectively communicating the full security status of user interactions is an open problem.

Phishing protection has been extensively studied in a web browser context (e.g., in [104, 105, 106]) and is commonly implemented using, for example, blacklists such as Google's SafeBrowsing [107]. Our work is complementary to these approaches and explores GUI confusion attacks that are not possible in web browsers.

In addition, the problem of presenting a trustworthy GUI has been studied and implemented in desktop operating systems, either by using a special key combination [108] or decorations around windows [109]. Given the limited amount of screen space and controls, applying these solutions in mobile devices would be impossible in an unobtrusive way.

Zhang et al. [79] show how UI attacks were extremely easy in Samsung devices (Samsung Galaxy S5 and S6) running Android 5. Specifically, these attacks were possible because, after a legitimate user touched the fingerprint reader sensor, a malicious app could use fingerprint-protected cryptographic keys generated by other apps. In addition, they show how in some devices it was possible to steal raw fingerprint information. In the work presented in Chapter 4, we focus primarily on the newer Google's fingerprint API (released with Android 6), in which each app utilizes app-specific keys.

Finally, a recent work shows how redressing attacks can even lead to complete compromise of the device UI [75], since an app can use these attacks to stealthy obtain "accessibility" permission and take full control of the user's input and the display's output.

5.2 Authentication-Related Vulnerabilities in Mobile Devices

Zhang et al. explore the issue of *uninstallation residue*, where uninstalling an app does not correctly clean up all data and references in the system, creating an opportunity for an attacker to elevate their privileges and steal sensitive information [110]. While the detection system we explained in Chapter 3 uses uninstallation of apps to trigger a removal of app-private data, the vulnerabilities targeted in this work and ours are very different. In fact, the system we described in Section 3.3 detects authentication vulnerabilities that involve the usage of both device identifiers and files in public locations.

The paper entitled Mayhem in the Push Clouds [111] explores the related issue of push messaging platforms, which are commonly used by apps to communicate asynchronously with their backends. The paper found that authentication tokens for these services are

often handled insecurely, especially when sent using Android's Intents. Our work focuses on what can happen when these tokens are created using device-public information, or are in turn stored as device-public information.

Chen et al., in OAuth Demystified for Mobile Application Developers [112], explore the usage of OAuth-like mechanisms for authentication. Their work included a manual study of 149 applications using OAuth and found that 89 used it incorrectly. Moreover, Wang et al. investigate OAuth misuse from a different angle in their paper Explicating SDKs [113], which examines the way applications use authentication and authorization SDKs from companies such as Facebook and Microsoft. A related, but distinct, vulnerability can occur in improperly implemented services using OAuth, which implicitly trust responses from identity providers without verification [114]. In contrast to web platforms, in mobile apps, these responses originate from the user being authenticated, meaning they can be tampered with, allowing an attacker to authenticate as the victim without their private credentials.

As we do in our work, Liu et al. [115] studies how apps unsafely use public storage. However, their work focuses on how the public storage is used to store sensitive information (such as the user's contact list), whereas we focus on how the public storage is used to store information that, together with device's identifiers, is used to authenticate with remote backends. Similarly, a work by Bai et al. [116] studies how a specific class of apps (backup tools) leaks information in publicly accessible files in the external storage. However, the apps studied by this work require either *root* or *shell* privileges, not obtainable by normal apps under the threat model we considered in Chapter 3 (non-compromised operating system).

Zuo et al. [92] developed a system, named AutoForge, to automatically find authentication vulnerabilities revolving around user-private information. Specifically, they focused on detecting apps' backends vulnerable to password brute-forcing, leaked user-

name and password probing, and Facebook access-token hijacking. We consider this work as complementary to ours. In fact, their work studies how apps' backend behaves when probed with supposedly-secret data, such as usernames, passwords, and Facebook authentication tokens. Conversely, our work focuses on an entire class of authentication schemes that do not rely on this supposedly-secret data.

A paper by Mulliner et al. [50] looks directly at the issue of SMS-based one-time passwords. They explore various layers of the problem, including issues of wireless interception, and smartphone Trojans, similar to our "ID Leaker." While their work was primarily motivated by the use of mobile Transaction Authorization Numbers in the banking industry, this same idea has also spread to most areas of the mobile world that require verification of a user's phone number, as we explore in our study. SMS authentication is further investigated by Schrittwieser et al. [117]. In this work, authors manually analyze a selection of messaging apps, verifying their security properties and finding different vulnerabilities in them.

The intrinsic weakness of SMS-based authentication has been recently pointed out. For instance, security researchers have shown that, by exploiting vulnerabilities of the SS7 network used by telecom company to route phone calls and SMS, it is possible for an attacker to intercept SMS and steal authentication codes [118]. Moreover, state-sponsored attackers could easily interfere with local telecom companies to intercept these authentication messages [70]. For this reason, the latest security guidelines advise against the use of SMS as a two-factor authentication method [119]. It is important to notice that the vulnerabilities we found in popular messaging apps (see Section 3.5.1) were *not* due to the usage of SMS content for authentication, but a consequence of the usage of publicly accessible files and device's identifiers to authenticate their users.

A few other works focus on aspects concerning the security of the fingerprint hardware sensor and the storage of fingerprint data [120, 121]. In the work presented in Chapter 4 we expand and generalize findings of these previous works and, in addition, we systematically study how apps use the fingerprint API and how this aspect affects the overall security of an app's authentication scheme.

There is a number of works related to two-factor authentication and authentication in mobile systems. Lang et al. [58] describe "Security Keys," second-factor devices that protect users against phishing and man-in-the-middle attacks. They also discuss the deployment of this technology to the Chrome browser and Google online services. In our work, we show how the fingerprint API could potentially offer the same or better security properties, but some shortcomings in its implementation and in how apps use it prevent this from happening.

One-Time Passwords (OTPs) are a (weaker) alternative to Security Keys. TrustOTP [86] shows how smartphones can act as secure OTP tokens. Dmitrienko et al. [68] highlight weaknesses in the design and adoption of two-factor authentication protocols and mechanisms. In particular, they show how an attacker can mount cross-device attacks and bypass 2FA mechanisms such as SMS-based TANs (Transaction Authentication Numbers) used by banks, or login verification systems such as Google, Twitter, and Facebook. Chen et al. [122] discuss different OAuth implementations and their adoption by mobile applications.

5.3 Automated Analysis of Mobile Applications

Rastogi et al. proposed AppsPlayground [41], a dynamic analysis framework aimed at maximizing code coverage of dynamic analysis. Other works with similar goals are Brahmastra by Bhoraskar et al. [123] and DynoDroid by Machiry et al. [124]. The vul-

nerability detection system described in Chapter 3 utilizes similar techniques to interact with apps, however, our goal is different, since we do not aim to maximize code coverage but to trigger the authentication mechanisms in a deterministic manner.

Different tools have been proposed to deterministically record and playback input events on Android: RERAN [125], MOSAIC [126], MobiPlay [127], and VALERA [128]. The usage of these tools as a part of our dynamic-analysis based vulnerability detection system constitutes an interesting future direction, since they could remove non-deterministic behaviors which currently hinder our analysis. However, in their current state, these tools do not completely solve the problem. For instance, RERAN, MOSAIC, and MobiPlay do not deterministically replay network traffic, whereas in our experiments we determined that most of the non-deterministic behaviors are due to discrepancies or delays in the network traffic between an app and its backend. The approach of VALERA can deal with network traffic, however, it cannot replay user's interaction in case of applications using customized rendering, like many of the ones we detected as vulnerable (see Section 3.5.2). Unfortunately, most of the apps we correctly detected as vulnerable actually use customized interfaces and heavily interact with online backends.

Several works focus on the automatic detection of classes of vulnerabilities in Android apps. Previous work focused on detecting over-privileged apps [129], component-hijacking vulnerabilities [130], vulnerable content providers [131], permission leaking [132], and vulnerabilities related to the unsafe usage of crypto-related APIs [91, 92], SSL connections [133], and dynamic code loading [134]. Other recent works show how some apps implement vulnerable custom authentication schemes (trying to minimize users' effort during login) [2] and how apps often use payment libraries insecurely [135]. Acar et al. provide an overview of the different security mechanisms implemented in Android and the improvements suggested by academia [136].

Chapter 6

Conclusions and Future Directions

In this thesis, I presented my studies on the current mobile ecosystem, how its components trust each other, and how an attacker can exploit these trust relationships.

It is important to notice that during my studies, some countermeasures to the identified security issues have been implemented in Android, in some cases with solutions similar to what we proposed. For instance, the behavior of the Android UI has been significantly changed over the years, progressively constraining how an app can control the device's UI, and giving more possibility to the user to limit app's capabilities.[137, 138, 139, 140]

More fundamentally, newer Android versions include mechanisms to ensure that an attacker cannot hinder user authentication, even when the standard, *untrusted*, operating system has been compromised. These recent additions constitute a radical change in the trust relationships between apps, remote backends, the operating system, and Trusted Execution Environments. We discussed some of these mechanisms in Chapter 4, but others have been proposed, and some of them will be implemented in the upcoming Android version ("Android P")[141, 142]. For instance, a secure UI mechanism, similar to what proposed in Section 4.8.1, will be available in this upcoming version.

While these new security mechanisms undoubtedly improve the security of the Android operating system, at this stage, it is unclear how devices' manufacturers will implement them and if apps' developers will use them correctly. Therefore, the primary future direction of this work will be to study *if* and *how* these mechanisms will be used, identifying common developers' mistakes and proposing modifications to the APIs used to control them, to ease their adoption.

Bibliography

[1] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, *What the App is That? Deception and Countermeasures in the Android User Interface*, in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[2] A. Bianchi, E. Gustafson, Y. Fratantonio, C. Kruegel, and G. Vigna, *Exploitation and Mitigation of Authentication Schemes Based on Device-Public Information*, in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2017.

[3] A. Bianchi, Y. Fratantonio, A. Machiry, C. Kruegel, G. Vigna, S. P. H. Chung, and W. Lee, *Broken fingers: On the usage of the fingerprint api in android*, in *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2018.

[4] comScore, "The U.S. Mobile App Report." http://www.comscore.com/Insights/Presentations-and-Whitepapers/2014/The-US-Mobile-App-Report, 2014.

[5] ESET, "Trends for 2013." http://www.eset.com/us/resources/white-papers/Trends_for_2013_preview.pdf.

[6] M. Niemietz and J. Schwenk, *UI Redressing Attacks on Android Devices*, *Black Hat Abu Dhabi* (2012).

[7] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du, *Touchjacking Attacks on Web in Android, iOS, and Windows Phone*, in *Proceedings of the 5th International Conference on Foundations and Practice of Security (FPS)*, (Berlin, Heidelberg), pp. 227–243, Springer-Verlag, 2012.

[8] A. P. Felt and D. Wagner, *Phishing on Mobile Devices*, in *Proceedings of the IEEE Workshop on Web 2.0 Security & Privacy (W2SP)*, 2011.

[9] Q. A. Chen, Z. Qian, and Z. M. Mao, *Peeking into Your App Without Actually Seeing It: UI State Inference and Novel Android Attacks*, in *Proceedings of the 23rd USENIX Security Symposium*, (Berkeley, CA, USA), pp. 1037–1052, USENIX Association, 2014.

[10] Bank of America, "SiteKey Security." https://www.bankofamerica.com/privacy/online-mobile-banking-privacy/sitekey.go.

[11] Yahoo, "Yahoo Personalized Sign-In Seal." https://protect.login.yahoo.com.

[12] E. Fernandes, Q. A. Chen, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash, "TIVOs: Trusted Visual I/O Paths for Android." University of Michigan CSE Technical Report CSE-TR-586-14, 2014.

[13] TrendLabs, "Tapjacking: An Untapped Threat in Android." http://blog.trendmicro.com/trendlabs-security-intelligence/tapjacking-an-untapped-threat-in-android/, December, 2012.

[14] TrendLabs, "Bypassing Android Permissions: What You Need to Know." http://blog.trendmicro.com/trendlabs-security-intelligence/bypassing-android-permissions-what-you-need-to-know/, November, 2012.

[15] S. Jana and V. Shmatikov, *Memento: Learning Secrets from Process Footprints*, in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pp. 143–157, May, 2012.

[16] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, *Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications*, in *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, (Berlin, Heidelberg), pp. 62–81, Springer-Verlag, 2012.

[17] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, *Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces*, in *Proceedings of the Second ACM Conference on Data and Application Security and Privacy (CODASPY)*, (New York, NY, USA), pp. 317–326, ACM, 2012.

[18] W. Zhou, X. Zhang, and X. Jiang, *AppInk: Watermarking Android Apps for Repackaging Deterrence*, in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS)*, (New York, NY, USA), pp. 1–12, ACM, 2013.

[19] P. De Ryck, N. Nikiforakis, L. Desmet, and W. Joosen, *TabShots: Client-side Detection of Tabnabbing Attacks*, in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS)*, (New York, NY, USA), pp. 447–456, ACM, 2013.

[20] Google, "Using Immersive Full-Screen Mode." https://developer.android.com/training/system-ui/immersive.html.

[21] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, *An Empirical Study of Cryptographic Misuse in Android Applications*, in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, (New York, NY, USA), pp. 73–84, ACM, 2013.

[22] A. Desnos and G. Gueguen, *Android: From reversing to decompilation*, *Black Hat Abu Dhabi* (2011).

[23] M. Weiser, *Program slicing*, in *Proceedings of the 5th international conference on Software engineering*, pp. 439–449, IEEE Press, 1981.

[24] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, *WHYPER: Towards Automating Risk Assessment of Mobile Applications*, in *Proceedings of the 22nd USENIX Security Symposium*, (Berkeley, CA, USA), pp. 527–542, USENIX Association, 2013.

[25] Y. Zhou and X. Jiang, *Dissecting Android Malware: Characterization and Evolution*, in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pp. 95–109, May, 2012.

[26] R. Unuchek, "The Android Trojan Svpeng Now Capable of Mobile Phishing." http://securelist.com/blog/research/57301/the-android-trojan-svpeng-now-capable-of-mobile-phishing/, November, 2013.

[27] CA/Browser Forum, "Guidelines For The Issuance And Management Of Extended Validation Certificates." https://cabforum.org/wp-content/uploads/Guidelines_v1_4_3.pdf, 2013.

[28] Google, "Featured, Staff Picks, Collections, and Badges." https://developer.android.com/distribute/googleplay/about.html#featured-staff-picks.

[29] J. Clark and P. van Oorschot, *SoK: SSL and HTTPS: Revisiting Past Challenges and Evaluating Certificate Trust Model Enhancements*, in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pp. 511–525, May, 2013.

[30] A. Kittur, E. H. Chi, and B. Suh, *Crowdsourcing User Studies with Mechanical Turk*, in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, (New York, NY, USA), pp. 453–456, ACM, 2008.

[31] StatCounter, "Operating system market share worldwide – may 2017." http://gs.statcounter.com/os-market-share#monthly-201705-201705-bar, 2017.

[32] Google, "AccountManager." https://developer.android.com/reference/android/accounts/AccountManager.html, 2016.

[33] Google, "Advertising ID." https://support.google.com/googleplay/android-developer/answer/6048248?hl=en, 2016.

[34] Google, "Google Play Developer Program Policies." https://play.google.com/about/developer-content-policy.html, 2016.

[35] S. Son, D. Kim, and V. Shmatikov, *What Mobile Ads Know About Mobile Users*, in *Proceedings of the 22nd Annual Network & Distributed System Security Symposium (NDSS)*, 2016.

[36] Google, "Using the External Storage." https://developer.android.com/guide/topics/data/data-storage.html#filesExternal, 2017.

[37] "Xposed Installer." http://repo.xposed.info/module/de.robv.android.xposed.installer, 2017.

[38] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna, *Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis*, in *Proceedings of the 24th Network & Distributed System Security Symposium (NDSS)*, February, 2017.

[39] Google, "Testing Support Library." https://developer.android.com/tools/help/uiautomator/, 2016.

[40] X. Cong, "uiautomator." https://github.com/xiaocong/uiautomator, 2015.

[41] V. Rastogi, Y. Chen, and W. Enck, *AppsPlayground: Automatic Security Analysis of Smartphone Applications*, in *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[42] I. A. Bureau, "Ad Unit Guidelines." http://www.iab.net/guidelines/508676/508767/ad_unit, 2015.

[43] J. Buchner, "Image Hash library." https://github.com/JohannesBuchner/imagehash, 2015.

[44] Google, "Platform Versions." https://web.archive.org/web/20160131030000/https://developer.android.com/about/dashboards/index.html, 2016.

[45] S. Demetriou, W. Merrill, W. Yang, A. Zhang, and C. Gunter, *Free for All! Assessing User Data Exposure to Advertising Libraries on Android*, in *Proceedings of the 23rd Network & Distributed System Security Symposium (NDSS)*, 2016.

[46] Google, "Implementing In-app Billing." https://developer.android.com/google/play/billing/billing_integrate.html, 2016.

[47] C. Mulliner, W. Robertson, and E. Kirda, *VirtualSwindle: An Automated Attack Against In-App Billing on Android*, in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (Asia CCS)*, 2014.

[48] C. Feng, *Playing with shadows – exposing the black market for online game password theft*, in *Virus Bulletin Conference*, 2008.

[49] A. Bianchi, "Implementation of the proposed defense mechanisms." https://github.com/ucsb-seclab/android_device_public, 2017.

[50] C. Mulliner, R. Borgaonkar, P. Stewin, and J.-P. Seifert, *SMS-Based One-Time Passwords: Attacks and Defense*, in *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2013.

[51] Google, "Android Documentation: SmsManager." https://developer.android.com/reference/android/telephony/SmsManager.html, 2016.

[52] Google, "Binder." https://developer.android.com/reference/android/os/Binder.html#getCallingUid(), 2016.

[53] Google, "Android O Behavior Changes." https://developer.android.com/preview/behavior-changes.html#privacy-all, 2017.

[54] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. 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)*, 2010.

[55] Board of Governors of the Federal Reserve System, "Consumers and Mobile Financial Services 2016." https://www.federalreserve.gov/econresdata/consumers-and-mobile-financial-services-report-201603.pdf, 2016.

[56] ARM, "ARM TrustZone." https://www.arm.com/products/security-on-arm/trustzone, 2017.

[57] Google, "Android Security Bulletins." https://source.android.com/security/bulletin/, 2017.

[58] J. Lang, A. Czeskis, D. Balfanz, and M. Schilder, *Security Keys: Practical Cryptographic Second Factors for the Modern Web*, in *Proceedings of the International Conference on Financial Cryptography and Data Security (FC)*, 2016.

[59] yubico, "YubiKeys." https://www.yubico.com/products/yubikey-hardware/, 2017.

[60] FIDO Alliance, "What is FIDO?." https://fidoalliance.org/about/what-is-fido/, 2017.

[61] Google, "New in Android Samples: Authenticating to remote servers using the Fingerprint API ." https://android-developers.googleblog.com/2015/10/new-in-android-samples-authenticating.html, 2015.

[62] A. Bianchi, "Source Code of the Developed Static Analysis Tool." https://github.com/ucsb-seclab/android_broken_fingers, 2018.

[63] J. V. Stoep, "Android: protecting the kernel." https://events.linuxfoundation.org/sites/events/files/slides/Android-%20protecting%20the%20kernel.pdf, 2016.

[64] Google, "Verifying Boot." https://source.android.com/security/verifiedboot/verified-boot, 2017.

[65] OWASP-MSTG, "Local Authentication on Android." https://github.com/OWASP/owasp-mstg/blob/master/Document/0x05f-Testing-Local-Authentication.md, 2017.

[66] Google, "FingerprintManager." https://developer.android.com/reference/android/hardware/fingerprint/FingerprintManager.html, 2017.

[67] Google, "Android Keystore System." https://developer.android.com/training/articles/keystore.html, 2017.

[68] A. Dmitrienko, C. Liebchen, C. Rossow, and A. Sadeghi, *On the (In)Security of Mobile Two-Factor Authentication*, in *Proceedings of the International Conference on Financial Cryptography and Data Security (FC)*, 2014.

[69] J. Scott-Railton and K. Kleemola, "London Calling – Two-Factor Authentication Phishing from Iran." https://citizenlab.ca/2015/08/iran_two_factor_phishing/, 2015.

[70] Telegram, "Keep Calm and Send Telegrams!." https://telegram.org/blog/15million-reuters, 2016.

[71] D. Goodin, "Thieves drain 2fa-protected bank accounts by abusing SS7 routing protocol." https://arstechnica.com/information-technology/2017/05/thieves-drain-2fa, 2017.

[72] yubico, "FIDO U2F." https://www.yubico.com/solutions/fido-u2f/, 2017.

[73] M. Niemietz and J. Schwenk, *UI Redressing Attacks on Android Devices*, *Black Hat Abu Dhabi* (2012).

[74] Q. A. Chen, Z. Qian, and Z. M. Mao, *Peeking Into Your App Without Actually Seeing It: UI State Inference and Novel Android Attacks*, in *Proceedings of the USENIX Security Symposium (Usenix SEC)*, 2014.

[75] Y. Fratantonio, C. Qian, P. Chung, and W. Lee, *Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop*, in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[76] A. Roy, N. Memon, and A. Ross, *MasterPrint: Exploring the Vulnerability of Partial Fingerprint-Based Authentication Systems*, *IEEE Transactions on Information Forensics and Security* **12(9)** (2017).

[77] Google, "Android FingerprintDialog Sample." https://github.com/googlesamples/android-FingerprintDialog, 2017.

[78] B. Parno, *Bootstrapping Trust in a "Trusted" Platform*, in *Proceedings of the USENIX Summit on Hot Topics in Security (HotSec)*, 2008.

[79] Y. Zhang, Z. Chen, and T. Wei, *Fingerprints On Mobile Devices: Abusing and Leaking*, in *Black Hat USA*, 2015.

[80] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, *Mining Apps for Abnormal Usage of Sensitive Data*, in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015.

[81] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, *Soot – a Java Bytecode Optimization Framework*, in *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, 1999.

[82] J. Palsberg and M. I. Schwartzbach, *Object-Oriented Type Inference*, in *Proceedings the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 1991.

[83] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, *EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework*, in *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2015.

[84] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, *Permission Evolution in the Android Ecosystem*, in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2012.

[85] Widevine, "Widevine DRM Architecture Overview." https://storage.googleapis.com/wvdocs/Widevine_DRM_Architecture_Overview.pdf, 2017.

[86] H. Sun, K. Sun, Y. Wang, and J. Jing, *TrustOTP: Transforming Smartphones into Secure One-Time Password Tokens*, in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[87] Google, "Material Design – Patterns – Fingerprint." https://material.io/guidelines/patterns/fingerprint.html, 2017.

[88] Google, "Android Key Attestation Sample." https://github.com/googlesamples/android-key-attestation/tree/master/server, 2016.

[89] Apple, "About Touch ID advanced security technology." https://support.apple.com/en-us/HT204587, 2015.

[90] Apple, "Get your apps ready for Touch Bar.." https://developer.apple.com/macos/touch-bar/, 2017.

[91] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, *An empirical study of cryptographic misuse in Android applications*, in *Proceedings of the 20th ACM SIGSAC conference on Computer & communications security (CCS)*, ACM, 2013.

[92] C. Zuo, W. Wang, R. Wang, and Z. Lin, *Automatic Forgery of Cryptographically Consistent Messages to Identify Security Vulnerabilities in Mobile Services*, in *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2016.

[93] F. Roesner and T. Kohno, *Securing Embedded User Interfaces: Android and Beyond*, in *Proceedings of the 22nd USENIX Security Symposium*, (Berkeley, CA, USA), pp. 97–112, USENIX Association, 2013.

[94] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, *Android Permissions: User Attention, Comprehension, and Behavior*, in *Proceedings of the Eighth Symposium On Usable Privacy and Security (SOUPS)*, (New York, NY, USA), pp. 3:1–3:14, ACM, 2012.

[95] R. Xu, H. Saïdi, and R. Anderson, *Aurasium: Practical Policy Enforcement for Android Applications*, in *Proceedings of the 21st USENIX Security Symposium*, (Berkeley, CA, USA), pp. 27–27, USENIX Association, 2012.

[96] Z. E. Ye and S. Smith, *Trusted Paths for Browsers*, in *Proceedings of the 11th USENIX Security Symposium*, (Berkeley, CA, USA), pp. 263–279, USENIX Association, 2002.

[97] A. Neupane, N. Saxena, K. Kuruvilla, M. Georgescu, and R. Kana, *Neural Signatures of User-Centered Security: An fMRI Study of Phishing and Malware Warnings*, in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[98] Y. Niu, F. Hsu, and H. Chen, *iPhish: Phishing Vulnerabilities on Consumer Electronics.*, in *Proceedings of the 1st Conference on Usability, Psychology, and Security (UPSEC)*, 2008.

[99] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor, *Crying Wolf: An Empirical Study of SSL Warning Effectiveness*, in *Proceedings of the 18th USENIX Security Symposium*, (Berkeley, CA, USA), pp. 399–416, USENIX Association, 2009.

[100] J. Lee, L. Bauer, and M. L. Mazurek, *The Effectiveness of Security Images in Internet Banking*, *Internet Computing, IEEE* **19** (Jan, 2015) 54–62.

[101] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. 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*, ACM, 2012.

[102] S. Schechter, R. Dhamija, A. Ozment, and I. Fischer, *The Emperor's New Security Indicators*, in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pp. 51–65, May, 2007.

[103] D. Akhawe and A. P. Felt, *Alice in Warningland: A Large-scale Field Study of Browser Security Warning Effectiveness*, in *Proceedings of the 22nd USENIX Security Symposium*, (Berkeley, CA, USA), pp. 257–272, USENIX Association, 2013.

[104] N. Chou, R. Ledesma, Y. Teraguchi, D. Boneh, and J. C. Mitchell, *Client-side defense against web-based identity theft*, in *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, 2004.

[105] R. Dhamija and J. D. Tygar, *The Battle Against Phishing: Dynamic Security Skins*, in *Proceedings of the Symposium On Usable Privacy and Security (SOUPS)*, (New York, NY, USA), pp. 77–88, ACM, 2005.

[106] E. Kirda and C. Kruegel, *Protecting users against phishing attacks with AntiPhish*, in *Proceedings of the Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 517–524 Vol. 2, July, 2005.

[107] Google, "Safe Browsing." http://www.google.com/transparencyreport/safebrowsing/.

[108] D. Clercq and Grillenmeie, *Microsoft Windows Security Fundamentals*. Digital Press, (Chapter 5.2.1), Connecticut, USA, October, 2006.

[109] J. Rutkowska, "Qubes OS Architecture (Section 5.3)." http://files.qubes-os.org/files/doc/arch-spec-0.3.pdf, January, 2010.

[110] X. Zhang, K. Ying, Y. Aafer, Z. Qiu, and W. Du, *Life after App Uninstallation: Are the Data Still Alive? Data Residue Attacks on Android*, in *Proceedings of the 23rd Network & Distributed System Security Symposium (NDSS)*, 2016.

[111] T. Li, X. Zhou, L. Xing, Y. Lee, M. Naveed, X. Wang, and X. Han, *Mayhem in the Push Clouds: Understanding and Mitigating Security Hazards in Mobile Push-Messaging Services*, in *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[112] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, *OAuth Demystified for Mobile Application Developers*, in *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[113] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich, *Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization.*, in *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*, 2013.

[114] R. Yang, W. C. Lau, and T. Liu, "Signing into One Billion Mobile App Accounts Effortlessly with OAuth2.0." BlackHat Europe, 2016.

[115] X. Liu, Z. Zhou, W. Diao, Z. Li, and K. Zhang, *An empirical study on android for saving non-shared data on public storage*, in *Proceedings of the IFIP International Information Security Conference*, 2015.

[116] G. Bai, J. Sun, J. Wu, Q. Ye, L. Li, J. S. Dong, and S. Guo, *All Your Sessions Are Belong to Us: Investigating Authenticator Leakage through Backup Channels on Android*, in *Proceedings of the 20th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2015.

[117] S. Schrittwieser, P. Frühwirt, P. Kieseberg, M. Leithner, M. Mulazzani, M. Huber, and E. R. Weippl, *Guess Who's Texting You? Evaluating the Security of Smartphone Messaging Applications*, in *Proceedings of the 19th Network & Distributed System Security Symposium (NDSS)*, 2012.

[118] Thomas Fox-Brewster, "Watch As Hackers Hijack WhatsApp Accounts Via Critical Telecoms Flaws." http://www.forbes.com/sites/thomasbrewster/2016/06/01/whatsapp-telegram-ss7-hacks/#43e6fc1c745e, 2016.

[119] NIST, "Digital Authentication Guideline." https://pages.nist.gov/800-63-3/sp800-63b.html, 2016.

[120] M. Rehman Zafar and M. Ali Shah, *Fingerprint Authentication and Security Risks in Smart Devices*, in *Proceedings of the International Conference on Automation and Computing (ICAC)*, 2016.

[121] T. Does and M. Maarse, "Subverting Android 6.0 fingerprint authentication." Master Thesis at University of Amsterdam, 2016.

[122] E. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, *OAuth Demystified for Mobile Application Developers*, in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2014.

[123] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall, *Brahmastra: Driving Apps to Test the Security of Third-Party Components*, in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, 2014.

[124] A. Machiry, R. Tahiliani, and M. Naik, *Dynodroid: An Input Generation System for Android Apps*, in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (FSE)*, 2013.

[125] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, *RERAN: Timing- and Touch-Sensitive Record and Replay for Android*, in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, 2013.

[126] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi, *Mosaic: Cross-Platform User-Interaction Record and Replay for the Fragmented Android Ecosystem*, in *Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015.

[127] Z. Qin, Y. Tang, E. Novak, and Q. Li, *MobiPlay: a Remote Execution based Record-and-Replay Tool for Mobile Applications*, in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016.

[128] Y. Hu, T. Azim, and I. Neamtiu, *Versatile yet Lightweight Record-and-Replay for Android*, in *ACM SIGPLAN Notices*, vol. 50, 2015.

[129] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, *PScout: Analyzing the Android Permission Specification*, in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[130] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, *CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities*, in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[131] Y. Zhou and X. Jiang, *Detecting Passive Content Leaks and Pollution in Android Application*, in *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2013.

[132] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, *Systematic Detection of Capability Leaks in Stock Android Smartphones*, in *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2012.

[133] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, *Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security*, in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[134] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, *Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications*, in *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2014.

[135] W. Yang, Y. Zhang, J. Li, H. Liu, Q. Wang, Y. Zhang, and D. Gu, *Show Me the Money! Finding Flawed Implementations of Third-party In-app Payment in Android Apps*, in *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2017.

[136] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith, *SoK: Lessons Learned From Android Security ResearchFor Appified Software Platforms*, in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2016.

[137] R. Whitwam, "Android O feature spotlight: Android tells you if an app is displaying a screen overlay." https://www.androidpolice.com/2017/05/21/android-o-feature-spotlight-android-tells-app-displaying-screen-overlay/, 2017.

[138] Google, "Android 6.0 Changes – Runtime Permissions." https://developer.android.com/about/versions/marshmallow/android-6.0-changes#behavior-runtime-permissions, 2015.

[139] J. Rummler, "Android Processes & Security." https://jaredrummler.com/2017/09/13/android-processes/, 2016.

[140] Google, "Android 8 Behavioral Changes – Alert windows." https://developer.android.com/about/versions/oreo/android-8.0-changes#all-aw, 2017.

[141] Google, "Android P Preview – Security updates – Android Protected Confirmation." https://developer.android.com/preview/features/security#user-confirmation, 2018.

[142] Google, "Android P Preview – Security updates – Hardware security module." https://developer.android.com/preview/features/security#hardware-security-module, 2018.