PRACTICAL LEAST PRIVILEGE FOR CROSS-ORIGIN INTERACTIONS ON MOBILE OPERATING SYSTEMS

BY

GÜLİZ SERAY TUNCAY

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

ABSTRACT

Mobile operating systems (i.e., mobile platforms) favor flexibility and re-usability as design principles and provide mobile applications with channels for establishing *cross-origin interactions* in an effort to realize these principles. Under this cross-origin scheme, applications accomplish tasks collaboratively with other principals, such as the web, other applications, and system components, by relying on them for certain functionality and data, considerably saving platform resources as well as the programming efforts of application developers. While clearly helping to provide a captivating mobile experience, this rich set of interactions within mobile platforms unfortunately also create significant attack vectors and pose notable security risks that need to be carefully taken into account to ensure the security of the platform.

In this thesis, we focus on the security of cross-origin interaction channels on mobile operating systems. We choose Android as a use case due to its popularity and open source and show that cross-origin interactions constitute a significant impediment to establishing *least privilege* on Android. More specifically, we show that there are severe issues in the security mechanisms that are put in place by Android or even a lack of adequate mechanisms to protect these interactions, which enable adversaries to stealthily obtain sensitive user data, high-risk platform resources, and application functionalities. We demonstrate the severity of these vulnerabilities by our measurement studies and showcases of exploits on popular real-world applications with millions of downloads on Google Play and show that a majority of Android users are rendered vulnerable due to these critical issues. As a remedy, we propose practical designs that strive to make Android more robust and secure by addressing the problems with cross-origin channels in a systematic, effective, and efficient manner.

In particular, we show that mobile applications that utilize embedded web browsers are under the risk of inadvertently exposing critical resources to untrusted web domains and we propose a systematic and practical access control mechanism to address this issue. Additionally, we disclose serious vulnerabilities in Android's permission framework that put inter-process communication and platform resources at severe risk and we redesign Android permissions to systematically resolve the issues. Finally, we present new attacks on Android's runtime permissions which proves that this new permission model cannot satisfy its key security guarantees due to design issues and vulnerabilities, jeopardizing the security of platform resources. We discuss the current approach taken by Android to mitigate these issues and demonstrate how this mechanism, although seemingly an ideal solution, is still intrinsically broken due to the existing vulnerabilities that we discovered.

All in all, this thesis aims to identify issues that threaten least privilege on mobile platforms and strives to provide practical mitigation solutions to resolve such issues. Our work has influenced the design and implementation of some of the critical security mechanisms in Android and has consequently led to changes in the official Android releases by Google. Even though we used Android as a use case here, the issues we disclosed in this thesis can also be encountered on other mobile platforms that utilize cross-origin interactions and our methodologies and designs go beyond Android to the design of mobile operating systems more generally.

To my parents Gülhan and Suavi and to the memory of my grandmother Ayşe.

ACKNOWLEDGMENTS

First, I would like to thank my Ph.D. advisor Dr. Carl Allen Gunter for his continuous guidance. Dr. Gunter gave me the freedom to work on problems that piqued my interest and thanks to this I discovered my passion in mobile security. He also provided invaluable ideas that greatly helped me improve my work. Being the humble and kind individual he is, he always knew how to give just the right feedback to motivate me. I also thank him for the fun times and giggles during one of our conference breaks, as we were navigating ourselves through the "strangely modern" artwork displayed at the museums of Vienna. I sure hope to do more of such adventurous museum visits with him in the future.

I would like to also thank my doctoral committee members Dr. Adam Bates, Dr. Suman Jana, and Dr. Tao Xie for their constructive feedback on my Ph.D. work. I especially thank Dr. Bates for his continuous encouragement, advice, and willingness to help on any academic or personal matter. I am also grateful to him for making an effort to arrange a comfortable office for me to help me complete my thesis without having my low back completely fail me.

I would like to thank all members of Security and Privacy Research at Illinois (SPRAI), especially Dr. Michael Bailey, Dr. Adam Bates, Dr. Nikita Borisov, Dr. Carl Gunter, Muhammad Haris, Wajih Ul Hassan, Deepak Kumar, Zane Ma, Dr. Andrew Miller, and Riccardo Paccagnella. It was a fantastic opportunity to get together with all these brilliant minds of computer security, who are always full of energy and ideas. I also thank them for all their feedback on my early research work that later led to conference publications. I feel privileged to be a part of such a great group and a positive environment during my Ph.D. at the University of Illinois at Urbana-Champaign (UIUC) and I will truly miss my interactions with them. I also thank my labmates from the Illinois Security Lab, Dr. Vincent Bindshaedler, Avesta Hojjati, Allyson Kaminsky, Hyun Bin Lee, Yunhui Long, Dr. Muhammed Naveed, Qi Wang, Dr. Wei Yang, and Dr. Aston Zhang, for sharing their stories and friendship.

I must express my gratitude to Dr. Ahmed Helmy, my M.Sc. adviser from the University of Florida (UF). He greatly supported me during my studies at UF and it was a great pleasure working with him. Thanks to Dr. Helmy's support, I started my Ph.D. at UIUC and he has been on my side throughout this journey.

I would like to also thank some of my friends I met during my time in Florida. Digvijay Singh Shaktawat, who unfortunately passed away earlier this year, was always there to support me and to cheer me up. It feels surreal to know such a genuine and kind person is no longer with us. I will forever miss him. I am also thankful to my friends Nilu Nalini, Giacomo Benincasa, and Damandeep Singh for their friendship, support, and for all the fun times.

I would like to extend my thanks to my professors, Adnan Yazıcı, Nihan Kesim Çiçekli, Halit Oğuztüzün, Ahmet Coşar, and Tolga Can, from Middle East Technical University (METU), for the great undergraduate education I have received and for always being very friendly and supportive. They are the ones who encouraged me to do my Ph.D. in the Unites States. I also would like to thank my friends from METU, especially Tolga Akın and Yıldız Dirmit for never leaving my side. I had some of the best times of my life at METU thanks to all these wonderful people and for that I am grateful.

I also had the privilege of forming some strong friendships in Urbana-Champaign. These friends have supported me immensely throughout this journey in different ways and I would like to thank them for all the great times, adventures, and their support in no particular order: Uğur Dönmez, Yıldız Yıldırım, Dr. Alperen Günay, Wajih Ul Hassan, Dr. Alessandro Contento, Dr. Minas Charalambides, Vasundhara Vigh, Shegufta Bakht Ahsan, Ally Kaminsky, Dr. Mehmet Kurt, Dr. Mo Dong, Zhenhuan Gao, and Muhammad Haris. They turned my time in Urbana-Champaign into a great experience and this town felt a bit closer to home thanks to them, though it is definitely very hard to beat my hometown İzmir, the pearl of the Mediterranean.

Finally, I owe my deepest gratitude to my family, who mean the world to me. I thank my brother Giray for his love and for setting a great example for me to also study computer science. I thank my mother Gülhan and my father Suavi, whom I should actually call Dr. Tuncay I, for supporting me at every step of this journey. Even though I have been through some hardships during my Ph.D., my parents were always there to share my pain and save me from my troubles. My mom was always on guard to pick up the phone, even at the wee hours of the morning, just to give me some comfort. At any point, one could see her eyes truly reflect my mental state; they shined bright with my happiness, their light withered if they witnessed my slightest sorrow. My dad fully believed in me at every point of my life and encouraged me to be a well-rounded person any way he could. He even wrote poems and composed songs for me if one can believe that. Lastly, I thank my dear grandmother Ayşe, for raising me up and for her endless love until her untimely passing away. She has unfortunately not been there to see most of my success story, but I know she would be so proud of her little Ayşe. I will remember her fondly until the end of my days. This thesis is dedicated to my family.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 OVERVIEW

Mobile operating systems place significant emphasis on flexibility and re-usability as design principles and promote mobile applications (apps for short) to rely on existing data and resources, in an effort to better utilize scarce platform resources (e.g., memory and storage) while also significantly reducing the programming efforts of app developers. To this end, mobile platforms offer a rich set of programming constructs that allow app developers to incorporate existing content into their apps from different origins, such as the web, other third-party apps, and the platform itself. Thanks to these *cross-origin interactions* [1], users can now enjoy seamless access to the web while using their favorite apps, they can smoothly utilize multiple apps to accomplish tasks, and they can gracefully utilize sensors and other platform resources to capture their surroundings and store their personal information on their devices. While these enhancements help apps gain notable revenue by creating an enthralling and engaging mobile experience for users, they unfortunately also create new attack vectors that can be easily exploited by adversaries. More specifically, any vulnerability in these interfaces can lead to adversaries escalating their privileges to perform high-risk tasks and obtain sensitive data, severely threatening the security and the privacy of mobile device users.

In this thesis, we choose the Android operating system as a use case due to its popularity (with over 80% of the market share) and open-source nature [2] and show that cross-origin interactions constitute a significant challenge to realizing the fundamental *principle of least privilege*, which states that each principal should be given the bare minimum privileges to perform its legitimate task. In particular, we take a deep dive into Android's platform internals to identify critical security issues that arise in cross-origin interactions due to the lack of fine-grained access control mechanisms or due to erroneous designs and implementations of existing access control mechanisms. We demonstrate the existence as well as extent of these problems with large-scale measurement studies on popular mobile applications and with showcases of real exploits on high-profile apps with millions of downloads and show that these issues indeed pose a real and significant threat to the security and privacy of billions of mobile device users. Recognizing the importance of systematically addressing security issues to prevent similar issues from resurfacing in the future, we carefully study the designs of the app components and security mechanisms that we identified to be problematic in order to determine the root causes of the issues. Based on this understanding, we propose practical access control solutions that strive to target the heart of the problems in an effective and efficient manner. *The security vulnerabilities we discovered in our work have officially been acknowledged by Google as serious threats that required swift attention and our work has consequently prompted critical changes in the official releases of Android, the most popular mobile operating system as of now.* It is worth mentioning that even though we focus on Android as a use case in this thesis, the vulnerabilities we discovered are not platform-specific and our methodologies and mitigation techniques go beyond Android to the design of mobile operating systems more generally.

Interactions between apps and web. Android provides cross-origin interactions between apps and web domains to bolster reusability on the platform and provide users with an overall well-rounded experience. Apps can integrate content from the web (i.e., web pages) through the use of a widely-used component called embedded web browser (viz., WebView) on Android. WebView helps developers minimize their programming effort by allowing them to reuse some of their existing web content and avoid reimplementing platform-specific app code. In an effort to provide a seamless interaction between apps and web domains, WebView provides programming bridges that can be used by app developers to tightly-couple their apps to certain web domains via allowing web code to access application code. We identified that the use of these programming bridges in mobile apps creates a serious attack vector as access to application code through the bridges is completely unregulated and any web domain can access the exposed resources, which creates an opportunity for web attackers to utilize the bridges to stealthily acquire app and device resources. Our investigation showed that Android provided very little security against such attacks and lacked the security mechanisms to allow app developers to regulate access to their app code and device resources. We demonstrate how a majority of the popular apps are vulnerable due to the lack of a fine-grained access model in WebViews and present case studies to show how popular apps can be attacked through the programming bridges to get access to sensitive user data (e.g., medical information). In order to systematically address this issue in WebView, we propose a fine-grained access control framework called Draco that allows app developers to easily express policy rules on how resources should be accessed by web origins via utilizing our easy-to-use and expressive policy language [3]. Draco requires no modifications to the Android source code as it is implemented in the Chromium System WebView app; hence it is easy to deploy without requiring OTA update of the operating system.

Interactions between different apps. In an effort to promote reusability and flexibility, Android also allows applications to communicate with and rely on each other for certain functionality and data. This is realized by allowing apps to expose some of their app components to other apps in order to accomplish certain tasks collaboratively via Android's built-in inter-process communication (IPC) mechanism, called the Binder IPC. In order to protect the exposed application components, Android introduces custom permissions, which is a security mechanism that is introduced as an addition to Android's permission system and is widely adopted by app developers. In this security scheme, third-party apps declare their *own* custom permissions and assign them to their components for protection from other apps. We discovered that this security mechanism is still far from achieving least privilege as it suffers from serious security vulnerabilities that enable adversaries to escalate their privileges on the platform and obtain unauthorized access to signature-protected components of apps and to high-risk device resources that require dangerous permissions (e.g., camera, microphone) [4, 5]. After carefully dissecting the design and implementation of this security mechanism, we discovered that at the root of these vulnerabilities lie overlooked design issues in custom permissions. In particular, the main problem, as we identified, is that custom permissions are treated and enforced the same way as system permissions by the Android platform, even though these two types of permissions are created by entities of different trust levels (i.e., system vs third-parties), consequently leading to privilege escalation attacks. To understand the extent of these problems, we conducted a large-scale measurement study on the most popular Android applications and showed that a majority of applications are rendered vulnerable due to these design issues in custom permissions. In addition, we demonstrated real exploits that target popular Android apps with millions of downloads and stealthily gain access to their private resources that have sensitive functionality, significantly challenging the security and privacy of device users. To systematically mitigate the issues, we propose a new model for Android custom permissions called Cusper, which provides an effective, efficient, and backward-compatible design that targets the root cause of the issues [6]. Furthermore, we built the first formal model of Android runtime permissions and used this model to formally verify the correctness of our approach. The security vulnerabilities we discovered in this work were acknowledged by Google as serious vulnerabilities that required prompt fixes and our work consequently led to changes in the future releases of Android [7, 8].

Interactions between apps and the system. Finally, Android allows apps to interact with the system to obtain certain platform resources (e.g., camera, microphone etc) and sensitive user data (e.g., SMS, contacts etc) in order to provide mobile device users a captivating experience. Access to these resources is regulated under Android's permission model where apps need to acquire the necessary permissions to obtain certain resources. Initially, all permissions were granted at installation time after the user agreed to install an app. However, this static model has received much criticism from the developer and security communities due to its lack of granularity, which only allowed apps to be granted either all or none of their requested permissions, and due to its lack of context, where users were not provided any explanation regarding the reasoning behind an app's use of permissions. In order to provide a remedy to this situation, Android 6.0 underwent a major update of its permission model and introduced runtime permissions, where apps would be dynamically granted dangerous permissions that are associated with high-risk resources (e.g., camera, contacts etc.), giving the user the opportunity to see in what context the app requested the permission as well as the chance to deny certain permissions. In our work, we identified that the correct operation of this model depends on certain implicit assumptions which are taken for granted by platform designers and adversaries can easily break these assumptions to obtain permissions from the background while impersonating foreground apps. This way, the security guarantees made by the runtime permission model are entirely broken as now an adversary can stealthily request permissions from the background without providing the user any real context and also they can obtain permissions on behalf of other apps, which is against Android's mandatory access control policy for permissions. We conducted a comprehensive user study to investigate if users' (lack of) understanding of runtime permissions would make them vulnerable to these attacks and also to understand if and how users could be led to fall victim to these attacks by utilizing their past behavior. Our results show that mobile device users are generally highly-vulnerable to our attacks and certain situations exacerbate their vulnerability. By utilizing our understanding from this user study, we build realistic phishing-based privilege escalation attacks on runtime permissions. Finally, we discuss the ideal defense strategy for phishing attacks in general and Android's current efforts on implementing this mitigation technique, which we unfortunately discovered to fall short due to vulnerabilities that can be utilized to bypass the mitigation. Hence, our attacks still work even in the latest Android releases that had promised to terminate phishing on the platform.

Thesis statement: Cross-origin interactions are a major source of vulnerabilities in establishing least privilege in modern mobile operating systems. However, the vulnerabilities can frequently be effectively addressed with well-chosen modifications in the design of the permission system or by introducing new access control mechanisms to the cross-origin channels in need.

1.2 THESIS CONTRIBUTIONS

In this thesis, we make significant contributions in the identification of serious design issues and security vulnerabilities in cross-origin interactions on real-world mobile operating systems as well as the design and implementation of practical mitigation techniques to resolve the identified issues. The issues we discovered pose major threat to millions of apps and billions of users and our designs strive to systematically mitigate the issues while being practical, effective, and efficient. Our contributions can be summarized as follows:

• *Identification of the overlooked lack of access control problem in embedded web browsers for all mobile web apps.* Previous work has shown that apps developed with hybrid frameworks that heavily rely on programming bridges in embedded web browsers to provide a cross-platform development experience are highly vulnerable to web attacks due to the lack of access control in these in-app browsers. We show that this problem is not limited to the scope of hybrid apps but in fact applies to all apps that utilize embedded web browsers. We demonstrate this by exploits on popular real-world apps that utilize embedded web browsers and show that adversarial web domains can stealthily obtain sensitive user data and device resources via the programming bridges.

• *Introduction of a fine-grained and uniform access control mechanism for embedded web browsers.* In order to provide protection against web adversaries, we built a uniform and fine-grained access control system for embedded web browsers to allow developers to specify their trusted web domains and the resources they trust each of these domains with for all types of apps. Our implementation resides in the Chromium system WebView app and does not require any modifications to the Android source code; therefore, it can be easily deployed without requiring any OTA update to the operating system.

• *Discovery and extensive analysis of serious vulnerabilities in Android custom permissions that lead to privilege escalation attacks on system resources, user data, and app components.* We discovered two new classes of attacks on custom permissions that allowed adversaries to escalate their privileges to stealthily obtain sensitive user data (e.g., contacts, SMS etc), high-risk platform resources (e.g., camera, microphone, location etc.), and private resources of other apps. These attacks are feasible to deploy in the real-world. Both of these attacks have been acknowledged by Google as serious security vulnerabilities and prompted Google to swiftly release security fixes to address the issues.

• *Identification of the root cause of the perpetual custom permission vulnerabilities.* After a detailed investigation of the design and implementation of Android permissions, we identified that the root cause of the perennial custom permission vulnerabilities is the system's inherent indifference to the trust levels of the creators of permissions. System permissions are declared by the trusted system, while custom permissions are declared by untrusted third-party applications; however, the Android framework does not distinguish between these two types of permissions at any point in time. This unavoidably leads to untrusted third-parties escalating their privileges to stealthily obtain high-risk system permissions and sensitive app components.

• *Implementation of a practical and systematic redesign of Android custom permissions.* We redesigned Android custom permissions in order to systematically resolve the perennial custom permission vulnerabilities via targeting the root cause of the issues. We separate the management of custom permissions from system permissions and add an internal naming scheme that allows us to track the ownership custom permissions to prevent spoofing attacks. Our approach is effective, efficient, and backward-compatible with third-party apps.

• *Introduction of the first formal model of Android permissions.* We built the first formal model of Android runtime permissions using the Alloy specification language and utilized this model in our formal analysis to prove the correctness of our mitigation of custom permission vulnerabilities. This formal model is a stand-alone contribution by itself as it can be used by other researchers to prove the correctness of other security properties or their designs and additions to the runtime permission model.

• *Discovery of serious vulnerabilities in runtime permissions that lead to phishing-based privilege escalation attacks.* Android runtime permissions strive to provide additional contextual information to the users to help them make informed decisions regarding permissions; however, we discovered that the security guarantees made by this model can be broken as adversaries can utilize the dynamic nature of runtime permissions to obtain permissions from the background while impersonating other apps. We built realistic attacks by studying the behavior of mobile device users and accordingly adjusting our attacks to target the situations in which users are highly vulnerable due to not fully understanding the intricacies of the runtime model.

• *Discovery of vulnerabilities in Android's recently-introduced mitigation against phishing attacks.* Android introduced a new mitigation approach in version 10.0 Q to resolve phishing attacks permanently by disallowing background apps to start activities. However, we found several ways to bypass this mitigation to still continue launching activities from the background.

• *Impact on the design and implementation of real-world mobile operating systems.* The issues we unearthed in this work have been acknowledged by Google as serious security vulnerabilities and led Google to update Android in order to promptly resolve the issues.

All in all, our work has impacted the design and implementation of Android, the most popular mobile operating system as of now.

• *Discoveries and new designs that go beyond Android to address vulnerabilities in cross-origin interactions on mobile operating systems.* The vulnerabilities we disclosed in this thesis are not platform-specific and can be found in other mobile operating systems that rely on cross-origin interactions. In addition, our research methodologies and designs can be generalized to other mobile operating systems as well.

1.3 THESIS ORGANIZATION

Chapter 2 presents the background concepts on the Android operating system and its security mechanisms. Chapter 3 discusses the related work on Android security. Chapter 4 focuses on protecting app to web interactions on Android. Chapter 5 presents the privilege escalation attacks we discovered on custom permissions, present our redesign of custom permissions to mitigate the attacks, and formal analysis of our mitigation for correctness. In Chapter 6, we present the phishing-based privilege escalation attacks we discovered on Android's runtime permission model and our strategies for producing realistic attacks. We then discuss existing mitigation strategies by Google and how we discovered they are broken due to security vulnerabilities. Finally, Chapter 7 concludes this thesis with a discussion of the applicability of higher-level mitigation strategies and future directions.

CHAPTER 2: BACKGROUND

2.1 ANDROID OPERATING SYSTEM

Android is an open-source mobile operating system released commercially by Google for the first time in 2008. Due to it being open-source, it is adopted by many major phone vendors (e.g., Samsung, HTC, LG etc.) and it is currently the most popular mobile OS as it has a large user base comprising of billions of users, simply dominating the mobile market share [9]. Starting from version 1.1, Android versions have traditionally been given dessert names that follow an alphabetical order (except for Petit Four), as can be seen in Table 2.1.

Table 2.1: Commercial Android versions [10]

Code name	Version number	Initial release date	API level
(No codename)	1.0	September 23, 2008	1
Petit Four	1.1	February 9, 2009	2
Cupcake	1.5	April 27, 2009	3
Donut	1.6	September 15, 2009	4
Eclair	2.0 – 2.1	October 26, 2009	5 – 7
Froyo	2.2 – 2.2.3	May 20, 2010	8
Gingerbread	2.3 – 2.3.7	December 6, 2010	9 – 10
Honeycomb	3.0 – 3.2.6	February 22, 2011	11 – 13
Ice Cream Sandwich	4.0 – 4.0.4	October 18, 2011	14 – 15
Jelly Bean	4.1 – 4.3.1	July 9, 2012	16 – 18
KitKat	4.4 – 4.4.4	October 31, 2013	19 – 20
Lollipop	5.0 – 5.1.1	November 12, 2014	21 – 22
Marshmallow	6.0 – 6.0.1	October 5, 2015	23
Nougat	7.0 – 7.1.2	August 22, 2016	24 – 25
Oreo	8.0 – 8.1	August 21, 2017	26 – 27
Pie	9.0	August 6, 2018	28
Android Q	10.0		29

2.1.1 Platform Architecture

Android is a Linux-based operating system; where the kernel is an optimized version or the Linux kernel with additional drivers implemented for IPC as well as for the sensors that are specialized for smartphones. On top of the kernel, the Android operating system provides libraries written in C, Android runtime for executing Java applications, and a Java application framework that can be used by app developers to implement their apps.

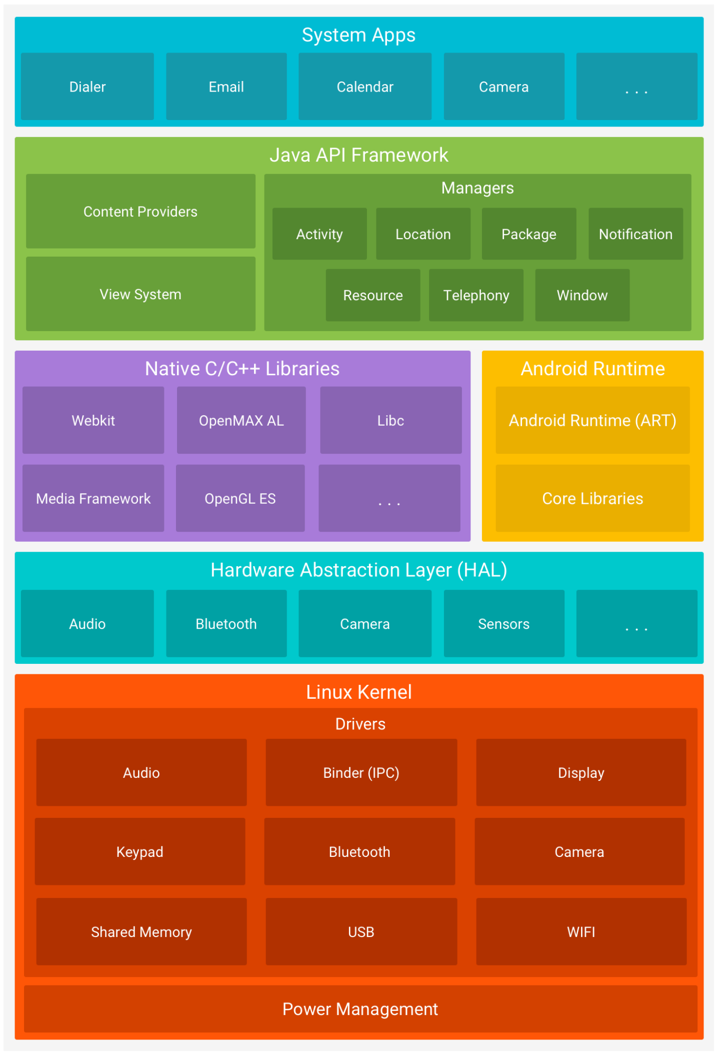


Figure 2.1: Android Software Stack [11].

Figure 2.1 illustrates the software stack of Android.

Android applications are typically written in Java by utilizing the Java API framework that simplifies the use of system components and services for developers. These apps are then compiled into byte code, which is later transformed into dex code by an Android IDE (typically Android Studio). Dex code is a more optimized and compact version of byte code specifically designed for mobile platforms to save storage space. In order to interpret and run dex code, Android platform also has a Java process machine. Previously, Android used Dalvik virtual machine, which performed Just-In-Time compilation to translate dex code into machine code. However, on Android 5.0, Android replaced Dalvik with Android RunTime (ART) which performs Ahead-Of-Time compilation at installation time in order to improve the runtime speed (particularly startup time) of apps and reduce their memory footprint. In its more recent versions, Android also provides Kotlin language support to app developers.

In addition, Android allows app developers to write parts of their apps in C in order to achieve better performance. This is done with the help of Java Native Interface (JNI), which is a programming framework that enables Java code running in a Java Virtual Machine (JVM) to call and be called by native applications. In order to utilize JNI, JNI's naming conventions should be followed when writing code in C.

2.1.2 Application Components

Android apps can comprise of four components: activities, services, broadcast receivers, and content providers. Each of these components (except content providers) can be an entry point to the apps. `Activities` are components that present the user with a graphical user interface to perform a single task. Activities follow a lifecycle, as can be seen in Figure 2.2. `Services` run in the background to perform long-running operations with no user interface. `Broadcast receivers` are used to receive broadcast messages from the system or the other apps. `Content providers` allow storing and sharing data between apps via a relational database interface. Communication between components is achieved through the Intent mechanism on Android. `Intents` are asynchronous messages between components in the same app or different apps that are used for activation of components.

Manifest file. Each Android app comes with a manifest file, which contains a set of declarations of app components, component capabilities, and app requirements. For each component declared in the manifest file, a certain permission can be assigned to protect the component and each component can be declared to receive specific types of messages (i.e., intents) from other apps (via intent filters). In addition, apps will declare their hardware requirements (e.g., sensors) and the types of permissions they need in the manifest file.

2.1.3 Inter-Component Communication

The Android operating system relies on inter-process communication (IPC)–also called inter-component communication (ICC) on Android–in order to achieve re-usability. Here, we present the details on how IPC is achieved and protected on Android.

Intents. Communication between components is achieved through the Intent mechanism on Android. `Intents` are asynchronous messages between components in the same app

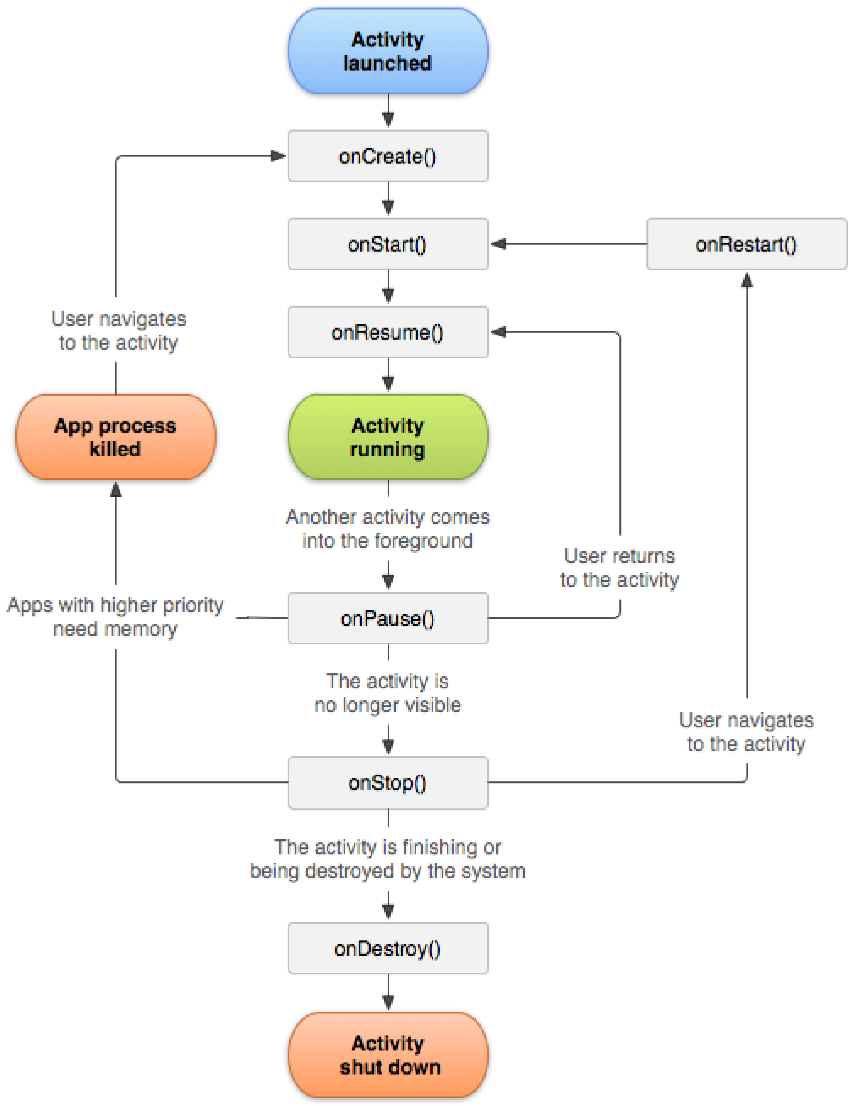

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 2.2: Activity Lifecycle [12].

or different apps that are used for activation of components. Intents contain a bundle of information: action to be taken on the receipt of the intent, data to act on, category of the component to handle the intent, and instructions on how to launch a target activity. Routing of intents can be explicit, where the intent is delivered to a specific receiver (i.e., component and package name will be specified) or, it can be implicit, where all of the components that registered to receive that action will be delivered the intent. In the manifest file, the developer can indicate which intents they want the components to respond to. This is done through the use of `intent filters`, where each component can declare the type of action, category, and data it is willing to handle.

Binder IPC. Unlike traditional Linux, for inter-process communication Android uses its own mechanism called the Binder IPC, which is implemented based on OpenBinder [13]. Compared to the traditional IPC mechanisms of Linux, Binder IPC provides optimized communication between processes for the Android platform. Binder IPC is implemented as a driver in the Linux kernel, where the kernel exposes the */dev/binder* device as an interface to enable IPC. The Binder driver is the central object of the Android framework which ensures that all IPC calls go through this driver.

2.1.4 Activity and Task Organization

A task is a collection of activities that are used to collaboratively perform a specific job. Android arranges activities forming a task into a back stack, in the reverse order of their start. Pressing the back button removes the top activity from the back stack and brings forth the activity underneath it to the foreground. In addition, recently-accessed tasks and activities can be obtained via the recents button that presents the system-level UI called the recents screen upon being clicked. Normally, the system handles the addition and removal of activities and tasks to the recents screen; however, this behavior can be overridden via some of the existing Android APIs. For example, `FLAG_ACTIVITY_NEW_DOCUMENT` flag can be passed to an `Intent` that is used for starting activity via the `addFlags()` API in order to add the activity to the recents screen. Additionally, activities can be launched as new tasks to be added to the recents screen by setting the `android:documentLaunchMode` attribute of the `<activity>` accordingly in the manifest file. Similarly, tasks can always be excluded from the recents screen by setting `android:excludeFromRecents` to true or by calling the `finishAndRemoveTask()` API in the activity that creates the new task.

2.1.5 Embedded Web Browsers

Android provides embedded web browsers (i.e., WebView) to allow apps to display web content fetched from the local storage or from the web. WebView is a widely-used component that forms the basic building block for modern web browser applications on the Android platform. Developers use WebViews to seamlessly integrate web content into their apps, without having to rely on a full-featured, heavy-weight web browser to render web content.

There are several interesting use cases for WebViews. For example, advertisement libraries use WebViews to display ad content within apps. In addition, app developers can rely on WebViews to tightly couple web sites with similar functionality to the app in order to reuse web site's UI code and to provide fast and convenient updates. Furthermore, hybrid frameworks such as PhoneGap rely on WebView's rendering capabilities to allow developers to write JavaScript and HTML code that can be easily ported to other mobile platforms.

2.1.6 WebView Implementation

WebView was first introduced in the API level 1 of the Android platform. It inherits from Android `View` and has additional rendering capabilities for displaying web pages. In Android 4.3 (JellyBean) and earlier, WebView implementation is based on Apple's WebKit browser engine [14], which powers several web browsers such as Safari, Google Chrome and Opera. WebKit consists primarily of two components: 1) WebCore, which provides the core layout functionality, and 2) JavaScriptCore, which is an engine that runs JavaScript. Starting from Android 4.4 (KitKat), Android dismissed the use of WebKit and the WebView implementation is instead based on Chromium [15], which is Google's widely-used, open-source browser project. Chromium uses Google's fork of WebKit, called Blink, as a rendering engine, and Google's high-performance V8 JavaScript engine.

Up until Android 4.4 (inclusive), the WebView implementation resided in the Android Open Source Project (AOSP) [2]; hence, any update to the WebView requires modifications to the operating system and can be pushed to users only with an OS update. With the introduction of Android 5 (Lollipop), WebView became a system app (called Android System WebView), presumably to ship updates quickly to the WebView code through Google Play. Apps that use WebViews load WebView code as a library into the app's process from the System WebView app.

2.1.7 WebView API

The WebView API allows app developers to load web content by calling the methods `loadURL()`, `loadData()`, `loadDataWithBaseURL()` and `postURL()` with a string argument that is the URL of the desired web content. JavaScript can be enabled on a WebView by calling `setJavaScriptEnabled()` on a `WebSettings` instance of a WebView. The source of JavaScript can be a file on the local storage or a remote domain. Additionally, the app can directly execute JavaScript by calling `loadURL()` with a string that starts with *"javascript:"* and is followed by the JavaScript code.

Navigation. Android developers have the option of controlling navigation within WebViews. Whenever the user clicks on a link in a page on a WebView, the developer can intercept this to make a decision on how this page should be loaded, or if it should be loaded at all. Developers have the option of allowing page loading from only certain domains, and

Listing 2.1: Navigation control in WebView

```
public boolean shouldOverrideUrlLoading(WebView view, String url) {
   URI obj = new URI(url);
   //allow alloweddomain.com pages to load in the WebView
   if (obj.getHost().equals("alloweddomain.com")) {
      return false;
   }

   //Load in browser
   Intent intent = new Intent(Intent.ACTION_VIEW), Uri.parse(url));
   view.getContext().startActivity(i);
   return true;
}
```

open pages from untrusted domains in the web browser. This can be implemented by overriding the `shouldOverrideUrlLoading()` callback method and checking the domain of the page before it is loaded.

Listing 2.1 shows how navigation control logic can be implemented. The developer implements `shouldOverrideUrlLoading` callback method to control navigation, allowing only content from "alloweddomain.com" to be rendered in the WebView, and everything else will be redirected to the system browser. Even though, this can help a great deal in protecting resources from untrusted origins, it is not enough to provide complete safety to applications since any trusted web site may include untrusted components such as an `iframe` for advertisements.

JavaScript interfaces. The WebView API allows inserting Java objects into WebViews using the `addJavaScriptInterface()` method. JavaScript loaded in the WebView can have access to application's internal Java code, giving web code the ability to interact more tightly with an app, and in some cases get access to system resources (e.g., hybrid frameworks). Mobile web apps commonly utilize JavaScript interfaces to meld web content with application code and provide users with a richer user experience compared to pure web apps.

Listing 2.2 shows how JavaScript interfaces can be used in applications. First, the app needs to register a Java object with a specific WebView instance and give this object a name. As shown in the example, this can be done by `addJavaScriptInterface(new MyJSInterface(),"InjectedObject")`. After this, JavaScript code running in the WebView can execute the methods of this object by using the name of the object and the name of the method, as in `InjectedObject.myExposedMethod()`.

Android API 17 introduced the use of `@JavaScript` annotation tag to export only the

Listing 2.2: JavaScript Interfaces in Android WebView

```java
mWebView.addJavaScriptInterface(new MyJSInterface(), "InjectedObject");
//...
public class MyJSInterface {

    @JavaScriptInterface
    public void myExposedMethod() {
        // do some sensitive activity
    }

    public void myHiddenMethod() {
        // JavaScript cannot access me, do some other activity
    }
}
```

desired Java methods of a Java class to JavaScript, primarily to prevent reflection-based attacks, where an adversary can use Java reflection to get access to the Android runtime and then execute arbitrary commands via calling `InjectedObject.getClass().forName("java.lang.Runtime").getMethod("getRuntime",null).invoke(null,null).exec(cmd)`. The use of the annotations is illustrated in Listing 2.2, where only the annotated method is made accessible to JavaScript. Even though API level 17 addresses a critical problem, it does not completely eradicate all the issues with WebViews. WebView still provides no access control on the JavaScript interfaces; any domain whose content was loaded into a WebView is free to use all the exported parts of the exposed Java object.

JavaScript event handlers. The WebView API allows developers to handle the `alert`, `prompt` and `confirm` JavaScript events, by registering the `onJsAlert()`, `onJsPrompt()` and `onJsConfirm()` Java callback methods, respectively. Whenever the JavaScript side calls any of these event methods, their respective handler will be called, if it is overridden. The developer is free to implement any logic in these event handlers. In fact, these event handlers are used in some hybrid frameworks to connect the web side to the local side.

In 2.3, the `alert` event handler checks if the string parameter starts with the desired label for this app. If this is the case, then the app parses the message and handles it according to its own logic. If the message does not start with the desired label, then the app falls back on the default behavior of alert() in JavaScript, which is simply displaying the message.

Handling HTML5 API requests. The rise of HTML5 has brought in a set of APIs that can give web applications the ability to access device hardware via JavaScript. Some

Listing 2.3: Overridden alert() event handler on Android

```java
public boolean onJsAlert(WebView view, String url, String message, JsResult
    jsResult) {
   if (message != null && message.startsWith("MyAppsLabel:")) {
      jsResult.confirm();
      parseMessageAndDoStuff(message);
      return true;
   } else { // falls back on the default behavior of alert() in JS
      return super.onJsAlert(view, url, message, jsResult);
   }
}
```

examples to these HTML5 APIs are `Geolocation` and `getUserMedia`, which enable access to GPS and to media devices such as camera and microphone, respectively. When a web domain requests access to one of these devices, the user should be prompted to grant access to this request. Starting from API level 21, Android WebView provides support for these HTML5 APIs and introduces mechanisms to grant or deny requests for accessing device hardware. In order to handle requests from web origins, the developer needs to make use of `onGeolocationShowPrompt` (for geolocation), and `onPermissionRequest` (for media devices) to grant or deny permission to the requests.

2.2 ANDROID SECURITY MODEL

Android incorporates different security mechanisms to ensure the security of apps and the platform as well as to provide users with appropriate privacy.

2.2.1 Application Sandbox

At its core, Android embraces the security mechanisms provided by the Linux kernel. It uses Linux's discretionary access control (DAC) mechanism to regulate access to the file system with the read, write, execute actions that can be granted to current user, group and public. Furthermore, Android implements process isolation, where a given user cannot interfere with another user's resources (files, memory, CPU resources etc.). Android uses these Linux security building blocks to implement its application sandbox; where each Android app corresponds to a different Linux user on the system and their resources including the processes, the files and CPU resources are protected from each other.

In Android, each app runs as a separate Linux user within its assigned sandbox with limited access to resources to ensure the integrity of the system and the apps. When an app wishes to use a resource outside its sandbox, it has to conform to Android's permission model and explicitly request it.

2.2.2 Application Signing

Digital certificates are the current standard to provide end-point authentication in today's communication paradigms. For example, on the web, when a client (e.g., web browser) communicates with a server (e.g., web domain), the server provides the client with a digital certificate that contains the public key of the server as a form proof of identity, which can then be verified by the client. This kind of authenticity verification relies on the establishment of a public key infrastructure (PKI), where trusted certificate authorities bind public keys to the identity of the entities in question.

Android also utilizes digital certificates as a proof of authenticity, which is necessary in order for apps to prove their authenticity to perform certain tasks. For example, in order to share a process with another app or to have a signature-protected form of IPC between multiple apps, Android requires the communicating apps to be signed by the same developer signature. In addition, in order for an app with a certain package name to get updates on the device, app's current signature has to match that of before to prevent repackaging attacks. In order to conform to Android's security model, every Android app has to carry a developer signature and certificate to be installed. However, as opposed to the web certificate ecosystem where a public key is tied to the *actual* identity of the respective entity (e.g., people and organization), Android utilizes a decentralized certificate model where certificates are self-signed by the app developers and serve as a trust-on-first-use (TOFU) identity. This is a design decision by Android framework developers to foster a free app market for developers where they can publish their apps quickly without running into many obstacles.

2.2.3 Android Permissions

In addition to the security features adopted from Linux, Android also implements other access control mechanisms to provide further security to the platform. First and foremost is the Android permission model, which aims to regulate access to system resources and sensitive user data by utilizing a set of security labels called permissions. By default, an Android application has access to only a limited set of system resources. In order to gain access, they have to request the necessary permissions in their manifest file with declarations of permission requirements. In contrast with the Linux's permission model for the file system, this permission model corresponds to a mandatory access control (MAC) mechanism, where all API calls go through a central reference monitor, which then verifies that a given user is granted the necessary permissions in order to successfully execute the respective API call. Camera, microphone, Internet, Bluetooth, SMS/MMS functions, location etc. are some of the resources that are protected via this permission model.

Each permission is associated with a protection level that indicates its severity. There are currently three main protection levels in Android: *normal*, *signature*, and *dangerous*. Resources that are associated with activities that present low risk to user's privacy (e.g., Internet) are protected with normal permissions; whereas high-risk resources (e.g., camera, contacts etc.) are protected with dangerous permission. Signature permissions protect private resources of apps, where the requester can be granted the permission only if it is signed with a matching certificate to that of the definer of the permission of interest.

Custom Permissions. Android ensures the security of inter-process communication (IPC) by utilizing its permission model. When two app components communicate, both the caller and the callee can require the other party to hold certain permissions for a successful communication. With this model, third-party apps are allowed define their own permissions. These third-party permissions, called *custom permissions*, are utilized by the system to regulate access to the app components. A component protected by a custom permission can be accessed through the Binder IPC only if the caller has the respective permission defined by the callee. In addition, the caller can also require the callee to hold certain custom permissions in order to be able tor receive messages from the caller.

In order to define a new custom permission, an app must provide a permission name and can optionally include a permission group to which this permission belongs and a description regarding the utility of the permission in its manifest as shown in the lines 2 - 7 of Listing 2.4. Additionally, in order to request a permission, an Android app needs to declare the use of the permission by referring to it with its name, as shown in line 10 of Listing 2.4. Furthermore, Android allows applications to create custom permissions dynamically via the use of the `addPermission()` API method. In order for this method to work successfully, apps need to declare permission trees in their manifest file which state the domain name under which the dynamic permissions will be created. Listing 2.5 demonstrates an example where a service is protected with a signature-level custom permission named `com.gulizseray.MY_PERMISSION` that was declared in Listing 2.4, other app components can be protected in a similar fashion.

Although it is suggested that reverse domain name notation should be used for custom permission names, there is currently no naming convention enforced by the system for custom

Listing 2.4: Creating and requesting custom permissions

```
 <!--Create a custom permission-->
<permission
    android:name="com.gulizseray.MY_PERMISSION"
    android:protectionLevel="signature"
    android:permissionGroup=
        "android.permission-group.STORAGE">
</permission>

<!-- Request a custom permission -->
<uses-permission android:name="com.example.PERM_NAME" />
```

Listing 2.5: Protecting a service with custom permissions

```
<service
   android:name=".MyService"
   android:enabled="true"
   android:exported="true"
   android:permission="com.gulizseray.MY_PERMISSION">
</service>
```

permissions and apps can use any name they desire when creating new custom permissions. One exception is that Android does not allow two different permissions to coexist on the same device if they have the same name; hence, installation of an app which defines a permission with a name that belongs to an existing permission on the device will be denied by the system. Conventional use case for custom permissions is for apps to define custom permissions with the signature protection level so that only the apps that are signed with the same certificate (e.g., apps that belong to the same developer) can utilize the resources of the definer app.

Runtime Permissions Model. Before Android 6.0 Marshmallow (API level 23), Android had been using a static permission model where permissions were granted to apps at installation time and they could never be revoked to apps unless the apps got uninstalled. Figure 2.3 shows how all dangerous permissions were previously displayed to the user at installation time on Google Play store. Previous work has investigated the effectiveness of this permission model and has concluded that users would benefit from having the ability to revoke permissions on demand [16]. Following these criticisms and after much demand from the security and user community, Android underwent a major change in its security model and switched to a dynamic permission model (i.e., runtime permissions) where permissions would be requested dynamically and could be revoked by the user at any time. The runtime

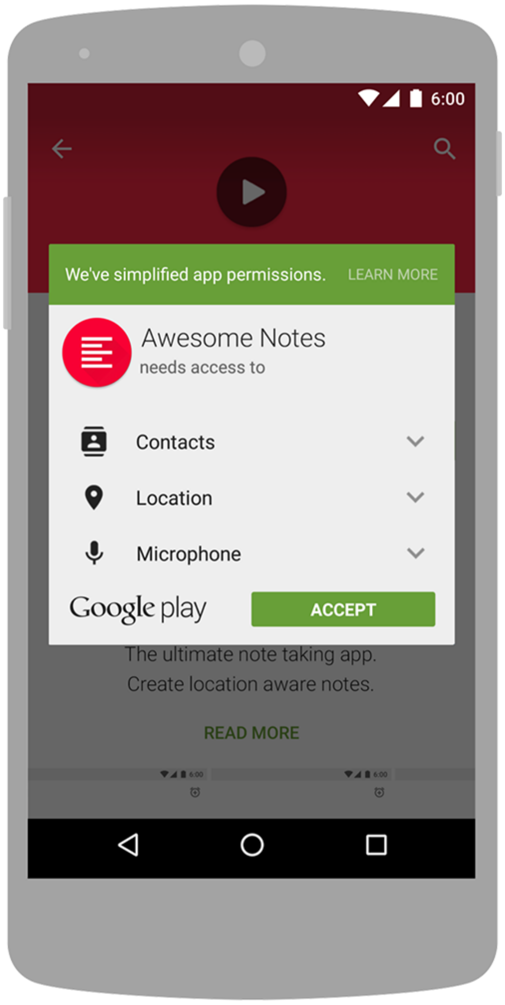
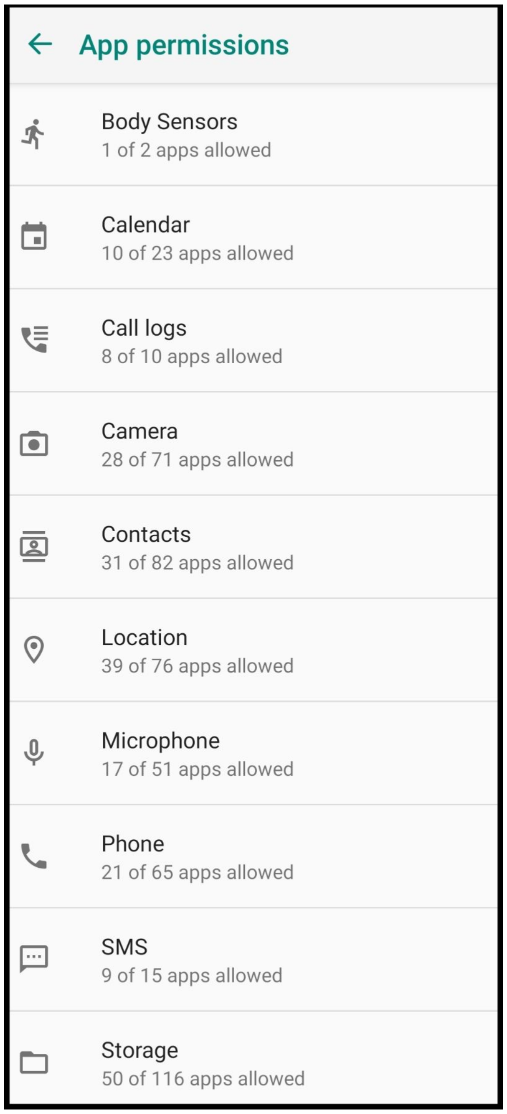


Figure 2.3: Permissions being displayed to the user during installation on Google Play [17].

permission model is a step towards achieving least privilege on Android, as it allows for a user-oriented model where apps will have to work only with a subset of their required permissions as seen fit by the device user.

Android runtime model cluster permissions into permission groups based on their utility [18]. In particular, there are 24 dangerous permissions in 10 permission groups on the latest version of the Android platform (Android 9.0 Pie), which protect access to high-risk system resources (e.g., storage, contacts, location, camera, microphone, SMS, sensors etc.). The full list of permission groups on Android 9.0 Pie can be seen in Figure 2.4. According to the runtime model, runtime permissions are granted on the basis of permission groups. If a dangerous permission in a permission group is granted to an app, all the dangerous permissions in that group will also be granted (if explicitly requested by the app) in order to minimize user's effort. Figure 2.5 shows how a runtime permission request looks like on Android.



App permissions

Body Sensors
1 of 2 apps allowed

Calendar
10 of 23 apps allowed

Call logs
8 of 10 apps allowed

Camera
28 of 71 apps allowed

Contacts
31 of 82 apps allowed

Location
39 of 76 apps allowed

Microphone
17 of 51 apps allowed

Phone
21 of 65 apps allowed

SMS
9 of 15 apps allowed

Storage
50 of 116 apps allowed

Figure 2.4: Permission groups on Android 9.0 (Pie).

In order to utilize runtime permissions, an app should have a target API level of at least 23 and it should run on an Android OS version that supports the runtime permission model. Legacy apps (i.e., target API level 22 or less) running on Android OS versions that implement runtime permissions will still adhere to the install time model; however, user is still given the ability to revoke permissions to the legacy apps through Settings and in this case the apps will likely crash if they did not correctly use security exceptions. Listing 2.6 shows how dangerous permissions can be requested and Listing 2.7 demonstrates how the result of a request (user granting/denying a permission) can be handled.

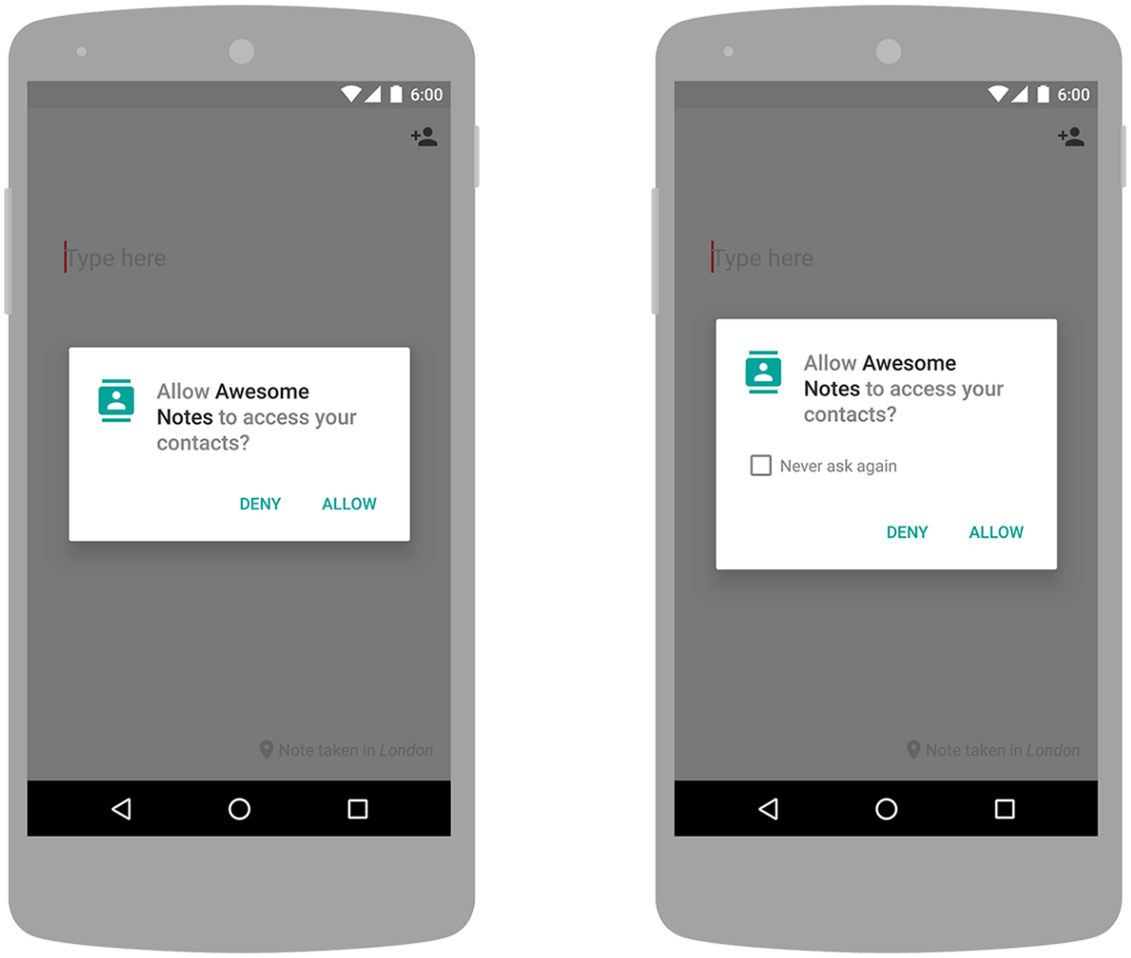


Figure 2.5: Runtime request for dangerous permissions. The dialog on the left is for initial permission requests whereas the dialog on the right is for secondary requests, which provide an option to turn off further requests [17].

2.3 RESEARCH METHODOLOGIES

2.3.1 Static Program Analysis

Static analysis is a methodology for testing and evaluating a piece of software by analyzing the source code without executing the software, in contrast with dynamic analysis, where the program is analyzed while it is being executed. The primary advantage of static analysis is that all possible execution paths can be examined statically and thus it can reveal errors that may not manifest themselves with dynamic analysis. Amandroid [20], Flowdroid [21], Androguard [22] are some of the popular static analysis frameworks designed for Android applications.

Listing 2.6: Requesting runtime permissions in Android [19]

```java
// Here, thisActivity is the current activity
if (ContextCompat.checkSelfPermission(thisActivity,
   Manifest.permission.READ_CONTACTS)
      != PackageManager.PERMISSION_GRANTED) {

         // Permission is not granted
         // Should we show an explanation?
         if (ActivityCompat.shouldShowRequestPermissionRationale(thisActivity,
            Manifest.permission.READ_CONTACTS)) {
               // Show an explanation to the user *asynchronously* -- don't block
               // this thread waiting for the user's response! After the user
               // sees the explanation, try again to request the permission.
         } else {
            // No explanation needed; request the permission
            ActivityCompat.requestPermissions(thisActivity,
               new String[]{Manifest.permission.READ_CONTACTS},
               MY_PERMISSIONS_REQUEST_READ_CONTACTS);

            // MY_PERMISSIONS_REQUEST_READ_CONTACTS is an
            // app-defined int constant. The callback method gets the
            // result of the request.
         }
} else {
// Permission has already been granted
}
```

2.3.2 Formal Verification (via Alloy)

Formal verification allows us to systematically reason about the design of complex systems by covering many cases that would otherwise be difficult to investigate with static analysis or software testing. In our work, we used the formal verification framework called Alloy, which is developed by a team led by Daniel Jackson at MIT [23]. Alloy is a declarative specification language that is used to model the behavior and structural constraints of complex systems [23]. It provides a modeling tool called Alloy Analyzer that operates based on first-order (i.e., predicate) logic and can be used to analyze formal models created with the Alloy language.

Alloy Language. Statements in Alloy can be interpreted both from object-oriented (OO) programming paradigm and from set theory perspectives. *Signature* is a declaration of a schema, which defines the vocabulary of the model. It is similar to the concept of a class in OO paradigm and to a set in set theory. It can consist of several *fields*, which are equivalent to fields in OO paradigm and to relations from a set theoretical perspective. *Facts* are global

Listing 2.7: Request callback for runtime permissions in Android [19]

```java
@Override
public void onRequestPermissionsResult(int requestCode,
   String[] permissions, int[] grantResults) {
      switch (requestCode) {
         case MY_PERMISSIONS_REQUEST_READ_CONTACTS: {
            // If request is cancelled, the result arrays are empty.
            if (grantResults.length > 0
               && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                  // permission was granted, yay! Do the
                  // contacts-related task you need to do.
            } else {
               // permission denied, boo! Disable the
               // functionality that depends on this permission.
            }
            return;
         }

         // other 'case' lines to check for other
         // permissions this app might request.
      }
}
```

constraints to the model that are always supposed to hold. *Predicates* define parametrized constraints, which can be interpreted as operations that can be performed in the model. *Functions* are expressions with declaration parameters and they return a result based on the parameters. *Assertions* are assumptions made on the model and they can be validated via the Alloy analyzer. Additionally, Alloy allows using multiplicity keywords as quantifiers in quantified constraints: `all` (universal quantifier), `some` (existential quantifier), `lone` (zero or one), `one` (exactly one), `no` (zero). Also, it is possible to use `some` (or `set` interchangeably), `lone`, and `one` for field declarations in signatures to indicate the number of elements a field can take and also for signature declarations to indicate the number of elements that can belong to the set of the signature.

Alloy Analyzer. The Alloy Analyzer tool performs only finite scope checks on the models. The analysis is sound since it can never return false positives and is complete up to a scope as the tool will never miss any counterexamples that are equal or smaller than the specified scope. As in traditional model checking, Alloy models are infinite, that is, the specification dictates how the components of a system should behave without any restrictions on their quantity. The analyzer also provides automated analysis by allowing automatic generation of examples that satisfy a given model as well as counterexamples to claims (i.e., assertions) that are expected to hold in the model.

2.3.3 Context-Free Grammars

A context-free grammar (CFG) is a notation used for describing the syntax of formal languages. It is defined a set of recursive rewriting rules that can generate all possible strings in a given language. A CFG consists of 1) a set of *terminals*, which are the characters of the alphabet of the language being defined, 2) a set of *non-terminals*, which are placeholders for patterns of terminals that can be generated via the nonterminals, 3) a set of *productions*, which are rules for rewriting nonterminals using other nonterminals and/or terminals. In computer science, the most common notation for describing CFGs is the Backus-Naur Form (BNF) notation [24].

CHAPTER 3: RELATED WORK

3.1 THIRD-PARTY CODE AND WEB ATTACKS

Previous work has discussed the problem that foreign code governs the same privileges as the host application in different contexts.

Protecting against third-party libraries and other inter-module threats. Third-party libraries governing the same permissions as the host app has been shown to be problematic by the previous work. Working towards solving this issue, AdSplit [25] suggests separation of ad components from the core app and running them in their own processes for protecting against the malicious activities that can be performed by potentially-malicious ad libraries. In [1], the authors discuss the vulnerabilities due to the nonexistence of origin-based protection on the Android system. More specifically, they show that third-party libraries make host apps vulnerable to cross-origin attacks on the app-to-app channels such as intent and the scheme mechanism. Their solution, Morbs, gives developers a means to express new policies about how two apps can communicate, and it labels messages between apps with their origins so that the developer-written permissions can be enforced at run time. One short-falling of this solution is that it works for only app-to-app communication channels. FlexDroid [26] gives developers a way of creating fine-grained access control policies on the system resources for third-party libraries based on Android permissions. To enforce the policies, they examine the Dalvik call stack at run time to identify the origin of the call and its associated permissions. Case [27] takes another approach and instruments apps with a module that can mediate access between the submodules (which can even be in the granularity of a Java class) of the app. These solutions are not limited to only app-to-app channels as Morbs and can protect an app against inter-module threats; however, they do not provide protection against arbitrary foreign content that can be loaded within a single in-app module (e.g., via web containers).

Analysis, attacks, and defenses for WebViews. Vulnerability of WebViews has been extensively discussed by previous work [28, 29, 30, 31, 32]. In [28], the authors present several classes of attacks that can be launched against apps that use WebViews. Chin et al. present a static analysis tool that can identify whether an app is vulnerable to WebView attacks [29]. Mutchler et al. present a large-scale analysis on mobile web applications, and present the trend of vulnerabilities in these applications. None of these work implement any defense mechanism targeting WebViews [30]. In [33], the authors present an access control mechanism for WebViews. Their approach uses static analysis to identify the use of security-sensitive APIs in the exposed Java class, and notifies the user if any such use is found. The user is then prompted to allow or completely block the binding of the Java object. The main drawback of this approach is that after the user allows the binding, they do not provide any origin-based access control, so all the origins still have the same access rights. Additionally, their focus is only on the permission-protected resources and they do not attempt to protect other application resources.

WebView-related attacks on hybrid frameworks and bringing origin-based access control. Georgiev et al. discuss the nonexistence of origin-based access control in hybrid frameworks and propose a capability-based approach (NoFrak), where app developer whitelists origins that are allowed to access system resources [34]. The drawback of their approach is that it works only for the Phonegap framework even though the aforementioned problem is not even specific to hybrid frameworks. Additionally, the solution is not fine-grained since a whitelisted origin get access to *all* resources of the app. In [35], the authors propose fine-grained access control system for hybrid apps, which allows developers to add origin permissions to the manifest file and associate iframes with permissions, and enforces the developer rules in the operating system. One drawback of this solution is that the web developer has to be compliant and include the permission tag along with the desired permissions in the iframes; otherwise, the frame just governs all the permissions the main page is given to. Furthermore, even though this solution provides a more fine-grained access control than NoFrak, it focuses on only protecting permission-protected resources, and hence is not enough to fully protect the app and its user as we have previously shown. Moreover, neither of these solutions give developers the flexibility to consult with the user on how to handle requests. In [36], the authors present code injection attacks on hybrid apps. Even though they mainly target hybrid frameworks, the attack shown can be applied to all mobile web applications in general.

Fixing Web-based system apps. Georgiev et al. show that Web-based system applications also suffer from similar problems, and introduce POWERGATE, which provides access control on native objects in the system by enforcing the policy rules created by the developer [37]. Here, their focus is on native-access APIs provided to the application by the platform, and not on the resources exposed by the use of JavaScript bridges.

3.2 ANDROID PERMISSIONS AND IPC

Previous work investigated Android Permissions and IPC security from many different perspectives.

IPC security on Android. Previous work has shown ways of exploiting IPC on Android to acquire unauthorized access to resources. In [38], the authors discuss the permission re-delegation problem where an unprivileged app can access system resources through a privileged app via IPC. Additionally, [39] shows ways of exploiting the Intent mechanism to send or receive Intents in an unauthorized manner and get access to other app's private resources.

Analysis of Android Permissions. Wei et al studied the evolution of permissions across Android versions and showed that the set of permissions on Android tends to grow with every release [40]. Stowaway tool aims to detect if apps follow the least privilege for permission requests [41]. Additionally, Bagheri et al present a formal analysis of Android permissions for older Android versions (<6.0) in Alloy [42]; whereas others introduce similar models in Coq [43, 44].

Android Runtime Permissions. One of the early works on runtime permissions shows the necessity of having revocable, ask-on-first-use type permissions on Android, supported by user studies [16]. [45] provides an initial analysis on the runtime permission model and identifies several problems in this model that might open up ways for exploits. In [46], the authors analyze the undesirable side effects of switching to runtime permissions and introduce a tool called RevDroid that aims to identify these problems in apps. DP-transform provides a tool which helps developers adapt to the runtime model by automatically introducing the permission requests required by the model into the application code [47]. In [48], the authors study the decision making progress of users in the runtime permission model by employing an in-situ user study. Their findings suggest that most important factor while granting or denying a permission is relevance of the permission to the app's functionality. In addition, user's trust in the app developer also appears to play a significant role in their willingness to grant runtime permissions.

Android Custom Permissions. Although previous work has studied Android permissions, there is little work done specifically regarding Android custom permissions.The blog post in [49] discusses how the "first one wins" approach for custom permission definitions can create problems. Shin et al presents a viable attack on custom permissions by exploiting the naming convention problem of custom permissions [50], to which Google responded with bug fixes. In [51], the authors discuss how permissions can stay dormant on the Android platform, later to be revived by the installation of a permission definer app, and demonstrate attacks on custom permissions via the exploitation of this undesirable property.

3.3 MOBILE UI SPOOFING ATTACKS

Mobile UI spoofing attacks on Android have been studied heavily by the security community. In these attacks, users are tricked into misidentifying an app and as a result they either inadvertently provide sensitive information to adversaries or they perform sensitive operations that will be beneficial for the adversaries without being aware of the consequences. Two main classes of UI spoofing attacks are phishing and clickjacking (also known as UI redress attacks). Bianchi et al. systematically studies the techniques that can be utilized to launch UI spoofing attacks and propose a defense mechanism that enables users to identify the authors of the apps they are interacting with.

Phishing attacks. In phishing attacks, the adversary surreptitiously replaces or mimics the UI of the victim to lead the user into thinking that they are interacting with the victim, while they are actually interacting with the adversary's app. Felt et al. discuss phishing attacks that can take place between mobile apps and websites on mobile platforms [52]. In [39], the authors discuss some of the intent vulnerabilities that can enable phishing attacks. More recently, Aonzo et al. discusses how a new app distribution mechanism on Android called instant apps can be utilized for phishing attacks in the context of password managers to trick them into supplying attackers with users' saved passwords [53].

In order to infer information about the foreground app, phishing attacks utilize either existing APIs or side-channels after these APIs were deprecated by Android in an attempt to prevent phishing attacks. In [54], authors discuss UI state inference attacks that rely on the proc filesystem (viz., procfs) to infer the UI state of apps (i.e., what is being displayed etc.) in real time. In [55], the authors use the interrupt channel on the procfs to infer the status of the foreground app. On more recent Android versions, procfs is highly restricted and the techniques we mentioned do not work anymore. More recently, Spreitzer et al. introduced ProcHarvester, which provides machine learning techniques to automatically identity the public resources in the procfs that can be utilized to infer the identity of the foreground app and keyboard gestures of the user [56]. Similarly, ScanDoid provides comparable utility using the Android APIs for the same purpose [57].

Clickjacking. In clickjacking, also known as the UI redress attacks, the adversary carefully places opaque and click-through overlays covering either parts of or the entirety of the victim app. While the user thinks they are interacting with the UI provided by the overlays, their clicks actually reach the victim and induce a state change in the victim that enables the attacker to achieve a sensitive task. In [58], Niemietz et al. demonstrate how UI redress attack, which are well-studied in the context of desktop-browser environments, can also applied to the Android platform. Wu et al. systematically study clickjacking attacks and propose a detection scheme to identify such attacks [59]. In [60], the attacker utilizes opaque overlays to lure the user into granting them the accessibility permission, which is an extremely powerful permission that allow the attacker to perform many sensitive tasks (i.e., installing apps, obtaining dangerous runtime permissions). Possemato et al. demonstrate that, contrary to common belief, not only system apps but also third-party apps suffer from clickjacking attacks and they propose a practical defense mechanism called CLICKSHIELD [61].

CHAPTER 4: SECURING WEB TO APP INTERACTIONS

Previous work has shown that web domains can obtain unauthorized access to critical app and device resources in apps developed with hybrid frameworks that rely on programming bridges to provide a cross-platform development experience. However, the main drawback of these existing works is that they considered the problem of excess authorization of web domains to be limited to these hybrid programming frameworks and proposed solutions that would work only within the context of these frameworks. In our work, we show that this problem in fact affects all applications–regardless of the programming framework used to build them– that utilize embedded web browsers and propose a general-purpose access control mechanism for web code in order to better address web overprivilege in Android. This chapter is based on our work in [3].

4.1 INTRODUCTION

Mobile application (or "app" for short) developers heavily rely on embedded browsers for displaying content in their apps and libraries. A previous study shows that 85% of the apps in the Google Play store contain at least one embedded browser (i.e., WebView on Android) [30]. Other than the natural use case of just displaying web content, there are some interesting ways to use these web containers in apps: advertisement libraries use embedded browsers to display ad content within apps, app developers can rely on embedded browsers to tightly couple web sites with similar functionality to the app in order to reuse web site's UI code and to provide fast and convenient updates. Additionally, hybrid frameworks (e.g., PhoneGap) rely on embedded browsers to enable app developers to write their apps purely with web languages (e.g., JavaScript and HTML) with the premise of ease of programming and portability to other mobile operating systems.

Even though they are extremely useful, these embedded browsers come with their own security problems. They are inherently given the ability to execute web code (i.e., JavaScript). Additionally, through the use of JavaScript bridges, they can allow web code to interact directly with app components (i.e., internal Java code). Indeed, these bridges are what hybrid apps rely on to allow access to system resources such as contact list, camera, Bluetooth, SMS etc. Obviously, the misuse of this functionality by malicious web domains can be detrimental to the user and to the app since an attacker, whose web domain (hence malicious code) was loaded into a WebView can exploit the existing bridges to collect information about the user and even change the app's behavior. The main problem here is that there is no means of performing access control on the untrusted code running within a WebView, any origin loaded into the WebView is free to use all the available JavaScript bridges. With the introduction of API level 17, Android made an attempt to mitigate the negative consequences of this problem (i.e., accessing Android runtime via Java reflection) by introducing mechanisms to allow the developer to specify which methods will be exposed to JavaScript. However, this does not eliminate the problem as the untrusted code loaded into a WebView still inherits the same permissions as the host app and can exploit just the exposed parts of the bridge to perform its malicious activities. Since the origin (as in same origin policy) information is not propagated through the bridge, the app developer has no control over this access attempt and cannot perform any access control based on the origin.

Prior research studies on security issues in WebViews and JavaScript bridges fall short in at least four significant ways. First, they have limited scope, since they mainly target hybrid apps and create solutions that work only for the hybrid frameworks [34]. Second, they are incomplete, since they focus only on protecting permission-protected resources (such as the camera and microphone) [34, 35], and disregard other cases where a foreign domain is inadvertently allowed to access sensitive information (such as a user's social security number). Third, they rely on whitelisting policies that always block unknown domains and therefore deprive developers of the flexibility to make decisions based on user input. Fourth, they are *ad hoc* since they focus only on a subset of resource access channels and do not provide a uniform solution that works across all channels.

The current disorganized and complex nature of interactions between web origins and applications creates confusion for developers. From our inspection of the apps in the Google Play store, we observed that the danger of loading untrusted web origins and exposing resources to them is not very well understood by app developers. Developers mistakenly assume that targeting API versions that address some of the issues with embedded browsers (e.g., using API level 17 or higher on Android) will protect their apps from these vulnerabilities. When they seem to be aware of the danger, assuring protection seems to be burdensome, and they tend to make mistakes while trying to evade the problem by implementing navigation control logic or multiple WebViews with different levels of exposure. However, even taking the correct programmatic precautions does not completely eradicate the problem since there is no guarantee that a trusted web domain will consist only of trusted components. Indeed, it is quite common for web pages to use an `iframe` in order to display ad content, and once loaded, there is no means for a developer to protect the resources that were exposed to web content from these potentially malicious components. All of this creates the necessity for an access control mechanism targeting web code where developers are given the ability to specify desired access characteristics of web origins in terms of app and device resources.

Developers should be allowed to specify what capabilities should be given to web origins with a fine granularity, and if they need user input to make decisions. This brings forth the need for a policy language, which developers can use to describe the expected behavior and use of resources by web origins, without having to rely on any complex programmatic structures, and the need for a mechanism that will take into consideration the developer policies to make access control decisions.

Working towards addressing these issues, we systematically study the vulnerabilities that are caused by loading untrusted web domains in WebViews on Android. We show cases where top-selling Android apps suffer from these vulnerabilities. Based on the threats we identified, we designed an easy to use, declarative policy language called Draconian Policy Language (DPL) for developers to specify access control policies on resources exposed to web origins. DPL allows declaration of policies with different levels of trust (i.e., fully-trusted, semi-trusted, untrusted) for different origins. We implement a system called Draco for fine-grained access control of web code: Draco enables app developers and device manufacturers (OEMs) to insert explicit Draconian policies into their apps, and dynamically enforces these policy rules at runtime in an efficient manner.

4.2 UNDERSTANDING THE PROBLEM

Embedded web browsers suffer from a lack of access control in Javascript bridges. In this section, we will elaborate this issue in detail and discuss how current security mechanisms in WebView fall short in providing the necessary access control. We will show how this issue affects a majority of popular applications on Google Play store and demonstrate case studies that show how two popular Android apps can be attacked through the JavaScript bridges to stealthily obtain access to sensitive user data (e.g., medical information).

4.2.1 Lack of Access Control in WebView

The vulnerabilities in WebViews have been investigated by previous work [30, 28, 29, 31, 32]. A recurrent and fundamental problem is that there is no way of performing access control on the foreign code executed within a WebView; any origin loaded into the WebView is free to use the exposed JavaScript bridges. In particular, since the origin information is not propagated to the app through the bridges, the app developer has no control over the behavior of foreign code and cannot make access decisions based on the real origin of the invocation. With the introduction of API level 17, Android addressed some critical problems of WebViews such as reflection-based attacks by introducing Java annotations into the WebView API to limit the extent of exposure. However, this does not completely solve the problem as the foreign code loaded into the WebView still has the same permissions as the host app, and it can exploit the exposed parts of the JavaScript bridges to perform malicious activities such as accessing system resources, getting the user's private information, and executing code that was meant for use only by the web domain of the developer.

In order for a JavaScript bridge to be exploitable, the app must load untrusted content into the associated WebView. An obvious way is by allowing the WebView to navigate to untrusted websites or to sites with untrusted content (e.g., `iframe`). Previous work shows that navigation to untrusted sites is common among applications: 34% of the apps that use WebViews do allow the user to navigate to third-party websites [30], and 42.5% of the apps that register a JS interface allow the user to navigate to third-party websites or to websites with untrusted content [29]. In order to verify these results, we picked three top-selling Android apps that demonstrate the common vulnerabilities identified by previous work: USPS, CVS Caremark, and JobSearch by Indeed. Through manually analyzing their code, we observed that developers do try to take precautions against the attacks on JS bridges by loading pages from untrusted domains in either the browser instead of the WebView (e.g., USPS app), or in separate WebViews with limited functionality which they create for this purpose (e.g., JobSearch app by Indeed). However, developers can make mistakes while implementing the navigation control logic. For example, in the USPS app, the developer checks if the loaded URL contains "usps.com" rather than checking if the host's domain name matches "usps.com", mistakenly allowing any non-USPS website that partially matches "usps.com" (e.g., musps.com, uusps.com). Additionally, developers might make wrong assumptions about the navigation behavior of the WebView. We have identified that the app developer might assume that the content provided to the WebView intrinsically does not allow navigation (i.e., it does not contain hyperlinks) and provide the user with functionalities that can break this assumption (e.g., allowing users to input hyperlinks) as in JobSearch app by Indeed, or they simply do not foresee that a specific WebView can be used by the user to navigate out of the trust-zone of the app by just following the links on the web pages as in the CVS Caremark app. We will examine the CVS Caremark and JobSearch apps in more detail later in this section.

Although it may look like correct implementation of navigation control would solve the JavaScript bridge exploitation issues (and fix the USPS app), we argue that it is simply an insufficient measure to protect JavaScript bridges. Even if developers implement all navigation behavior correctly and do not allow the user to navigate to untrusted web origins within the context of their apps, the pages from trusted domains might include untrusted components such as `iframe`s, which also inherit the same permissions as the app and have access to all the exposed bridges. Thus, the system does not provide the necessary means for developers to completely protect their apps against attacks on the JavaScript bridges.

4.2.2 Prevalence

While the current design of JavaScript bridges by default grants access to a domain for all the exposed resources, for HTML5 APIs the access model is exactly the opposite; the default behavior is to deny all the requests by a domain for a permission unless the `onPermissionRequestResult`, `onPermissionsRequest`, and `onGeolocationPermissionPrompt` methods are overridden by the app developer. In order to better understand how this difference between the JavaScript bridges and the HTML5 APIs affects the developers, we statically analyzed the top 1337 free Android applications from 21 Google Play categories selected at our discretion. Table 4.1 depicts the prevalence of WebViews and how often WebView APIs are used in these apps. Here, we distinguish ad and core WebViews (WebView in the core of the app) based on our comprehensive list of package names for ad libraries, and also give the cumulative result including both uses of WebView. In line with the previous work [30, 29], we have observed that WebView is a commonly-used component as around 92% of the applications in our dataset make use of it in their core application code (i.e., not used by advertisement libraries). Among the applications that include at least one WebView in their core code, 77% of them use JavaScript interfaces, and 70% use event handlers. However, it can be observed that there is a sudden drop in the numbers when it comes to the use of HTML5 APIs. This might be happening for two reasons. On the one hand, developers generally do not wish to grant access to permission-protected resources to external domains, and apps can operate without having to rely on external web origins. On the other hand, even though more than 85% of the apps in our dataset target API 21 or higher and are able to handle HTML5 API requests, they simply decide not to do so, possibly due to the complex request handling logic of the HTML5 APIs. Since the majority of the apps require API 21 or higher, they need to comply with the run time permissions introduced by Android 6.0. This means that each time they wish to grant access to a web domain, they also need to check if the app was granted the permission of interest by the user and, if not, they must prompt the user to grant it. On top of this, they also need to implement origin-based access control; hence, they need to maintain the necessary data structures and track user preferences. This process is unnecessarily cumbersome. If the system provides the necessary infrastructure to allow developers to uniformly declare their security policies, this would minimize effort and reduce the likelihood of errors, irrespective of the underlying channel (whether that is a JavaScript

Used Web Features	# (%) in core	# (%) in ad	# in both core and ad	Total # (%)
WebView	1226 (92%)	551 (41%)	544 (41%)	1233 (92%)
JavaScript enabled	1189 (89%)	495 (37%)	482 (36%)	1202 (90%)
JavaScript Interfaces	945 (71%)	395 (30%)	361 (27%)	979 (73%)
@JavaScriptInterface	587 (44%)	328 (25%)	182 (14%)	769 (58%)
onJsPrompt	857 (64%)	202 (15%)	172 (13%)	887 (66%)
onJsAlert	696 (52%)	259 (19%)	227 (17%)	923 (69%)
onJsConfirm	699 (52%)	214 (16%)	184 (14%)	883 (66%)
onGeolocationPermissionsShowPrompt	567 (42%)	169 (13%)	133 (10%)	700 (52%)
onPermissionRequest	32 (2%)	0 (0%)	0 (0%)	32 (2%)

Table 4.1: Prevalence of WebViews and use of WebView APIs (#: absolute number, %: percentage)

interface, an event handler or an HTML5 API).

4.2.3 Case Studies

Previous work has shown how applications built with hybrid frameworks suffer from JavaScript bridge vulnerabilities since hybrid frameworks rely on these bridges to give application code access to device resources like the camera, contact list and so on [34, 35]. However, the problem is not constrained to hybrid applications. In fact, an application that uses an embedded browser to display web content and needs enhanced communication between the web domain and the app, is susceptible to similar exploits. Here, we investigate the understudied JavaScript bridge issues that exist in non-hybrid applications. In particular, we present our analysis on two widely-deployed applications distributed through the Google Play store, which we found as suffering from JavaScript bridge issues. Indeed, we show that the exposure of these bridges to adversaries can be detrimental to users' privacy and can adversely affect the application's flow. We have disclosed these issues to the developers of those apps but—at the time of writing—haven't received a response.

CVS Caremark. CVS Caremark is one of the Android apps offered by the American pharmacy retail company CVS. It has been downloaded 100,000 times so far and currently has a rating of 3.6. It allows users to track their prescription history, get refills or request mail service for new prescriptions, and get information about drugs and their interactions. In order to help their users with their medical needs, the app requires them to register to the CVS system with their name, health care ID, and email address. The app additionally tracks some other personal information, including the user's pharmacy preferences and location. Furthermore, the app implements some functionality to check the login state of users, retrieve some internal database IDs, perform UI functionality such as displaying date pickers, and invoke the browser to load a given URL.

CVS Caremark app uses WebViews to render web content, and utilizes JavaScript interfaces to enable a tight communication between the app and CVS web servers. It registers two different interfaces with the WebView, of classes `WebViewJavascriptInterface` and `JavaScriptWebBridge`, and names the JavaScript objects associated with these interfaces *"native"* and *"WebJSInterface"* respectively. We observed that the *"native"* interface implements some of the main functionalities of the app (e.g., scanning prescriptions, registering users etc.) with access to device resources and exposes user's private information. Hence, it is highly possible that this interface was meant by the developer to be used for internal use only, that is by trusted CVS domains. On the contrary, the *"WebJSInterface"* is possibly meant to be used in a more generic context and by untrusted domains; hence, it exposes functionalities more conservatively. However, this attempt to protect the app and device resources by creating two interfaces is not useful, since both of these interfaces belong to the same WebView instance, which is used to load all URLs. Hence, the security of the app relies on implementing navigation control correctly, by not allowing the app to navigate to untrusted domains or to domains that might contain pages with untrusted elements. The app simply does not make any attempt to mitigate the problem by implementing navigation control to filter untrusted domains; hence, it is vulnerable to JavaScript bridge attacks. Even if navigation control was implemented correctly, this would not be enough to protect from these attacks since even a trusted origin may include elements that are of risky nature (e.g., `iframe`). We have successfully performed an attack on this JavaScript interface bridge by navigating to our attack URL which runs the code in 4.1. The attack domain was able to retrieve personal information (like the user's name, health care ID, email address, pharmacy preference, and location) as well as execute app functions such as using the camera on the victim device for barcode scanning functionality and retrieving images of user's prescription drugs.

Job Search by Indeed. Job Search is an app released by Indeed, a company that produces an employment-related search engine (*indeed.com*), to allow users to search for jobs on their Android devices. The app is downloaded 10,000,000 times and has a rating of 4.1.

Most of the content displayed to the user in the app is fetched from Indeed's web domain (*indeed.com*) and rendered in a WebView. This is done mainly to reuse the UI code of Indeed's web domain in order to reduce the app development effort and simplify maintenance of its deployment. Similarly to the CVS Caremark app, JobSearch creates and uses two types of WebView classes, one (of class `IndeedWebView` that extends the `WebView` class) for internal use and another (of class `ExternalWebView` that extends `IndeedWebView`) for

Listing 4.1: JavaScript interface exploitation in CVS Caremark

```
  function beEvil() {
     deviceInfo = native.getDeviceInfo()
     clientID = native.getBenefactorClientInternalId()
     geolocation = native.getGeoLocation()
     loginState = native.getLoginState()
     userName = native.getUserName()
     preferredPharmacy = native.getPreferredPharmacy()
     native.scanRx() // scan barcodes
     // get prescription barcode image
     prescriptionImage = native.getFrontRxImgData()
     data = constructData(deviceInfo, clientID, geolocation, loginState,
        preferredPharmacy, userName, prescriptionImage)
     // Send data to server
     b=document.createElement('img')
     b.src='http://123.***.***.***/?data='+ data
     native.setPreferredPharmacy("WhicheverPharmacyIWant!")
}
```

showing external content such as job descriptions from untrusted domains. The app attaches a JavaScript interface (named *"JavaScriptInterface"*) to the internal WebView, while not exposing any such interfaces to the external one in an attempt to protect resources from external domains. The app also takes precautions to restrict navigation in the internal WebView by removing all the hyperlinks in the rendered text content; hence, it supposedly does not allow loading of external URLs in this WebView. However, the app also offers the user the choice of adding web sites as a part of their profile and allowing the user to navigate to these sites. This breaks the developer's no-load assumption on the internal WebView for pages from external domains, and thereby puts the exposed JavaScript interfaces at risk. In fact, we were able to access the JavaScript interface by navigating to our "malicious" website in the internal WebView. In this interface, the app offers the device's unique ID, enabling/disabling Google Now, checking if this device is registered with Indeed, getting user's registration ID, and registering the device with Indeed.

4.2.4 Adversary Model

Based on our observations on how JavaScript bridge vulnerabilities can be exploited, we assume a web adversary who owns web domains and/or ad content in which he can place malicious code. Mobile web apps render the malicious domains and malicious ad content through their embedded browsers. Such an app can reach a malicious domain when the user starts navigating through the embedded browser. Moreover, malicious ad content can be offered by the adversary to both the app's trusted domains and to other untrusted domains. We assume that the adversary can reverse engineer the victim app code to identify the exploitable JavaScript bridges. The adversary can achieve this by first downloading the victim app's apk using existing frameworks for crawling Google Play store, and then decompiling it with any of the existing dex decompilers (dex2jar [62], JD-GUI [63], apktool [64] etc.).

4.3 DRACO ACCESS CONTROL

Running code from untrusted origins in a WebView can be detrimental to users as the foreign code can compromise users' privacy and disturb their experience by exploiting the WebView's tight-coupling with the application code and device resources. A straightforward way to address this threat is to simply prevent the user from visiting untrusted web pages. However, there are cases where this is impossible to achieve since even trusted domains might embed untrusted components in their web pages. Hence, it is necessary to provide an access control mechanism for WebViews that can distinguish the source of foreign code that is being executed and grant access only if the source is trusted by the developer or by the user. To tackle this problem, we propose *Draco*, a fine-grained, origin-based access control system for WebViews, which consists of two major components: 1) an access control policy language that we call the *Draconian Policy Language* (DPL), which allows app developers to declare policy rules dictating how different components within a WebView should be exposed to different web origins, and 2) a runtime system we call *Draco Runtime System* (DRS), which takes policy rules on system and internal app resources (i.e., JavaScript bridges) as an input from the developer and enforces them dynamically when a resource request is made by a web origin.

4.3.1 Design Goals

Before we go into the depths of our policy language and the Draco runtime access control on WebViews, it is important to discuss what our design goals are and how they affect the architecture of our system. Previous work focuses on access control only on the permission-protected parts of the exposed bridges in hybrid frameworks. Our goal is to provide developers with a fine-grained access control model, which will enable them to express access control policies on *all* parts of *all* access channels for *all* use cases of WebViews (i.e., in hybrid and native apps). These channels are namely the *JavaScript interface*, the *event handlers*, and the *HTML5 API*. App developers should be given full control on all of the channels, that is, they should be able to specify which origins can access which parts of the channels and assign permissions to trusted origins. They should also be given the flexibility to delegate decisions to the user when needed. Draco should avoid modifications on any parts of the operating system and should be implemented as part of a userspace app. This would allow the system to be readily and immediately deployable, while enabling frequent updates that are disjoint from firmware updates. Additionally, Draco should be able to enforce policy rules efficiently and its policy language should be easy to understand and use for developers, as well as easy to extend if necessary.

4.3.2 Draconian Policy Language

Draco supports a declarative policy language that allows app developers to describe their security policies with respect to remote code origins. Here we present the *Draconian Policy Language* (DPL) and provide examples to demonstrate its expressiveness.

Grammar. We want to instantiate a capability-based access control scheme based on least privilege and allow specification of what resources each remote origin can access and how they can access them. By default, if a DPL rule does not exist to allow the web code to access any resource, then access is denied. We use the Backus-Naur Form (BNF) notation [24] for context-free grammar to describe the new policy language. Terminals are denoted by single-quoted literals.

Draco allows developers to write policy rules which dictate how sensitive resources can be accessed by web code. We define the syntax of a DPL rule as:

$$\langle \textit{policy rule} \rangle ::= \langle \textit{subject} \rangle \text{ `}\texttt{;}\text{' } \langle \textit{trust level} \rangle \mid \langle \textit{subject} \rangle \text{ `}\texttt{;}\text{' } \langle \textit{channel} \rangle \text{ `}\texttt{;}\text{' } \langle \textit{decision point} \rangle$$

Each Draconian policy rule is applied on a subject. The *subject* indicates the web origin whose web content was loaded in the WebView. Here, a remote origin is represented by a URI scheme (i.e., http or https), a hostname and a port number as in the same origin policy. We allow wild cards for origins in our language, in order to allow creation of rules that can be applied to any origin. Additionally, we allow wild cards for sub-domains in domain names (e.g., (*).mydomain.com) to enable rules that can assign all hosts under the same domain the same access characteristics.

$$\langle \textit{subject} \rangle ::= \text{`}\texttt{*}\text{'} \mid \langle \textit{protocol} \rangle \, \langle \textit{hostname} \rangle \, \langle \textit{port} \rangle$$

$$\langle \textit{protocol} \rangle ::= \text{`}\texttt{http://}\text{'} \mid \text{`}\texttt{https://}\text{'} \mid \varnothing$$

$$\langle \textit{hostname} \rangle ::= \langle \textit{subdomain} \rangle \ \langle \textit{domain name} \rangle$$

$$\langle \textit{domain name} \rangle ::= \text{string}$$

$$\langle \textit{subdomain} \rangle ::= \text{`}\texttt{(*).}\text{'} \mid \langle \textit{name} \rangle \ \text{`}\texttt{.}\text{'} \mid \varnothing$$

$$\langle \textit{name} \rangle ::= \text{string}$$

$$\langle \textit{port} \rangle ::= \text{`}\texttt{:}\text{'} \ \langle \textit{port number} \rangle \mid \varnothing$$

$$\langle \textit{port number} \rangle ::= \text{integer}$$

A *trust level* is an abstraction that allows developers to instantiate default policies. Our system supports three trust levels:

$$\langle \textit{trust level} \rangle ::= \text{`}\texttt{trustlevel}\text{'} \ \text{`<'} \ \langle \textit{trust level options} \rangle \ \text{`>'}$$

$$\langle \textit{trust level options} \rangle ::= \text{`}\texttt{trusted}\text{'} \mid \text{`}\texttt{semi-trusted}\text{'} \mid \text{`}\texttt{untrusted}\text{'}$$

A *trusted* subject is allowed to access all resources whereas an *untrusted* subject is never allowed to access any resources through any channels. A *semi-trusted* domain can access exposed functionality but only through user interaction. At this point it should be clear that our policy language allows for essentially whitelisting domains. However, this is still not expressive enough. Consider for example the case that we want to allow a subject to access the exposed JavaScript interfaces but not run HTML5 code that can unilaterally access resources. Towards this end, the second part of the DPL rule definition allows for such fine-grained declarations. In particular an app developer can specify which *channel* should be protected. A *decision point* dictates whether such a policy rule should be enforced transparently to the user or only when the user agrees to it. If left empty, then "system" is assumed which forces the system to enforce the rule transparently to the user. If "user" is chosen then the system delegates the enforcement decision to the user at the time of the access attempt. DPL also allows app developers to provide a description message for the user. This can be useful in cases the DPL rule governs resources at a very fine-granularity (e.g. at the method level) which might be challenging for the user to understand. In such cases a semantically meaningful message provided by the app developer could help the user better perceive the context.

$$\langle \textit{decision point} \rangle ::= \text{`}\texttt{decisionpoint}\text{'} \ \text{`<'} \ \langle \textit{decision maker} \rangle \ \text{`>'} \ \langle \textit{description} \rangle$$

$$\langle \textit{decision maker} \rangle ::= \text{`}\texttt{system}\text{'} \mid \text{`}\texttt{user}\text{'} \mid \varnothing$$

$$\langle \textit{description} \rangle ::= \text{`}\texttt{<}\text{'}\ \langle \textit{text} \rangle\ \text{`}\texttt{>}\text{'} \mid \varnothing$$

$$\langle \textit{text} \rangle ::= \text{string}$$

The *channel* definition is more intricate: Draco needs to allow greater levels of rule expressiveness to enable developers to dictate fine-grained policies. Every channel has its own idiosyncrasies and exposes resources in different ways. This obviates the need for allowing different specifications for each channel. Thus, an app developer should be able to choose the channel they want to protect:

$$\langle \textit{channel} \rangle ::= \langle \textit{event handler} \rangle \mid \langle \textit{html5} \rangle \mid \langle \textit{jsinterface} \rangle$$

Our access control follows a least privilege approach: by default everything is forbidden unless there is a rule to allow something to happen. In particular, for the `event handler` channel, our policy allows app developers to specify how the whole channel can be accessed by the subject, but also—need be—to define which permission-protected APIs can be utilized by each event handler method. This is reflected in the policy language as follows:

$$\langle \textit{event handler} \rangle ::= \text{`}\texttt{alloweventhandler}\text{'}\ \text{`}\texttt{;}\text{'}\ \text{`}\texttt{<}\text{'}\ \langle \textit{eh methods} \rangle\ \text{`}\texttt{>}\text{'}\ \text{`}\texttt{;}\text{'}\ \text{`}\texttt{<}\text{'}\ \langle \textit{permission list} \rangle\ \text{`}\texttt{>}\text{'}$$

An event handler method list (*eh method list*) can be a single event handler method, or a list of event handler methods. Furthermore, developers should be allowed to specify a list of permissions for the exposed event handler methods:

$$\langle \textit{eh methods} \rangle ::= \text{`}\texttt{all}\text{'} \mid \langle \textit{eh method list} \rangle$$

$$\langle \textit{eh method list} \rangle ::= \langle \textit{eh method} \rangle \mid \langle \textit{eh method} \rangle\ \text{`}\texttt{,}\text{'}\ \langle \textit{eh method list} \rangle$$

$$\langle \textit{eh method} \rangle := \text{`}\texttt{onJsHandler}\text{'} \mid \text{`}\texttt{onJsPrompt}\text{'} \mid \text{`}\texttt{onJsConfirm}\text{'}$$

$$\langle \textit{permission list} \rangle ::= \langle \textit{permission} \rangle \mid \langle \textit{permission} \rangle\ \text{`}\texttt{,}\text{'}\ \langle \textit{permission list} \rangle \mid \varnothing$$

where a *¡permission¿* can be any of the Android permissions.

Similarly, for the *html5* channel one can specify the WebKit permissions that web code can make use of:

$$\langle \textit{html5} \rangle ::= \text{`}\texttt{allowhtml5}\text{'}\ \text{`}\texttt{;}\text{'}\ \text{`}\texttt{<}\text{'}\ \langle \textit{HTML permission list} \rangle\ \text{`}\texttt{>}\text{'}$$

$$\langle \textit{HTML permission list} \rangle ::= \langle \textit{HTML permission} \rangle \mid \langle \textit{HTML permission} \rangle\ \text{`}\texttt{,}\text{'}\ \langle \textit{HTML permission list} \rangle$$

$\langle$*HTML permission*$\rangle$ ::= '`VIDEO_CAPTURE`' | '`AUDIO_CAPTURE`' | '`GEOLOCATION`' | '`PROTECTED_MEDIA_ID`' | '`MIDI_SYSEX`'

Lastly, for the *jsinterface* channel, our policy language allows developers to describe how every Java class and Java method exposed to JavaScript can be accessed by the subject:

$\langle$*jsinterface*$\rangle$ ::= '`allowjsinterface`' '`;`' $\langle$*class methods*$\rangle$ '`;`' '`<`' $\langle$*permission list*$\rangle$ '`>`'

$\langle$*class methods*$\rangle$:= $\langle$*class name*$\rangle$ '`<`' $\langle$*methods*$\rangle$ '`>`'

$\langle$*class name*$\rangle$ ::= string

$\langle$*methods*$\rangle$ ::= '`all`' | $\langle$*method list*$\rangle$

$\langle$*method list*$\rangle$ ::= $\langle$*method name*$\rangle$ | $\langle$*method name*$\rangle$ '`,`' $\langle$*method list*$\rangle$ | ∅

$\langle$*method name*$\rangle$ ::= string

Expressiveness. As with any language, there exist an intrinsic tradeoff between the usability of the policy language and its expressiveness. On the one hand, a usable policy language is of low complexity but at the same time limited on the policies it can express. On the other hand, a complex policy language can express more fine-grained rules. DPL strikes a careful balance between the two by aiming to express selected set of useful policies at the method level with one-line, concise rules.

Consider for example the case where a developer of a low risk application would like to allow their web service (*"mydomain.com"*) to run code within the WebView of the host app. In such cases, the developer could simply provide a rule as follows:

```
https://mydomain.com;trustlevel<trusted>
```

Given only this rule, the system forbids any web code of origin other that *"mydomain.com"* to access any exposed functionality from the host app. At the same time, the trusted *"mydomain.com"* can benefit from all the exposed features.

In the aforementioned vulnerable case of the CVS Caremark app (see Section 4.2), it is evident that the app developers wanted to allow the CVS domains to access a rich JavaScript interface (*WebViewJavascriptInterface*) and other domains to access a more conservative *JavaScriptWebBridge* interface. This could be simply described by the app developer and enforced by DRS providing the following DPL rules:

```
https://www.caremark.com;allowjsinterface;
    WebViewJavascriptInterface<all>;decisionpoint<system>

*;allowjsinterface;JavaScriptWebBridge;decisionpoint<user>
```

where "*" is a wildcard that can match any origin. The former rule allows only CVS domains to access the sensitive APIs, whereas the non-sensitive app functionality can be exposed to all domains after user approval with the latter rule.

In the case of the "Job Search" app by Indeed, the developer could simply provide the rule:

```
(*).indeed.com;allowjsinterface;JavaScriptInterface; decisionpoint<system>
```

This will allow only code from *"indeed.com"* to use the exposed JavaScript interfaces. The developer does not need to worry about implementing two different WebViews, one for secure domains and one for untrusted domains. Furthermore, navigation will not be an issue as the system transparently allows only the *"(*).indeed.com"* domains to access sensitive APIs.

In fact, we identified by looking at the decompiled application code that the "Job Search" developers have written around 550 lines of code, aiming to achieve separation between trusted and untrusted domains by using a second fully-developed WebView (along with custom-built Activity, WebViewClient, WebChromeClient classes), and yet the app was still vulnerable. In contrast, one line of code with the Draconian Policy Language is enough to secure the app with Draco. Additionally, even though DPL allows the construction of very fine-grained policies, it can be seen from these examples that simple and easy-to-construct policy rules can be sufficient in many practical cases. Consequently, our system can provide strong protection to apps and minimize developers' efforts with an easy to use policy language.

MyStore example. To demonstrate more fully the expressiveness of DPL, we consider a more elaborate scenario. Let us assume that *MyStore* is a large retail company that aims to incorporate Eddystone [65] Bluetooth Low Energy (BLE) [66] beacons on product shelves in its stores [67]. These beacons broadcast URLs for the product they advertise using the BLE protocol. *MyStore* also provides its clients with a shopping app, namely *MyStore App*, which scans for the BLE beacon advertisement messages and displays the web page of the advertised products in a WebView. The advertised websites provide further information about the product such as description, images, reviews etc. These web pages can belong to the web domain of *MyStore* (*mystore.com*), or to the *MyStore* suppliers that partner with

MyStore to use store beacons. *MyStore App* collects a user's profile and preferences, allows her to scan product barcodes, and also acquires the location of the user's mobile device to perform analytics in order to better their services.

MyStore App is a mobile web application that uses components from *mystore.com* via the help of the WebView embedded browser, and exports device resources and app functionalities through Java Script bridges. In particular, it exposes the `MyInterface` Java class, which features the following functions: `getAge()`, `getGender()`, `get StoreLocation()`. *MyStore* wants to allow *mystore.com* to access all the JavaScript Interfaces and resources of *MyStore App*. At the same time, it wants to allow its partners' web domains to access the location of the store for them to know where their products are being seen. Furthermore, it wants to provide its partners the opportunity to get the user's age and gender. However, providing this information might entail privacy concerns. Thus, the store wants to allow their partner to access this information only if the user agrees, which will not violate the user's expectations and thus minimize the privacy risk. Enforcing this complex interactions can be extremely challenging with the current state of affairs on the Android platform, since it might require implementing multiple WebViews with different levels of exposure and duplicating method definitions. However, it becomes much easier when Draco is used. Consider for example the following Draconian policy rules:

```
mystore.com;trustlevel<trusted>

partner.com;allowhtml5;permission<GEOLOCATION>

partner.com;allowjsinterface;MyInterface<getLocation>; decision<system>

partner.com;allowjsinterface;MyInterface<getAge,getGender>;
    decision<user><"Access to age and gender">
```

The first rule allows *mystore.com* to access all exposed resources on all channels. The second and third rule allow the trusted *partner.com* domain (e.g., partner companies) to access the location through either HTML5 APIs or exposed interfaces of the *MyStore App*. Finally, the last rule allows *partner.com* to access the exposed functions that provide the user's age and gender only if the user agrees. As can be seen, writing policy rules using DPL should not require much effort from the app developers .

4.3.3 Draco Runtime System

Draco's other major component is its runtime system, namely the *Draco Runtime System* (DRS). A key aspect of the design of DRS is to avoid any modifications in the Android OS to make DRS easier to both deploy and update. In particular, DRS is built on top of the WebView system app on Android. This requires making modifications only in the Chromium source code [15], which is the provider of the WebView implementation on the Android platform.

High-level architecture. Keeping our design goals in mind, we implemented DRS in Chromium [15], which is a system library residing in the Android WebView system app, providing the WebView implementation to the Android apps. Figure 4.1 illustrates our design in more detail. The app developer provides DPL rules to the WebView programmatically through their Android app. DRS features a Policy Manager class (i.e., `PolicyManager`), which parses the policy rules and inserts them into a policy map data structure. We also implemented a unit for decompiling the app and statically analyzing it to determine the permissions necessary to successfully execute the methods in exposed Java class methods and event handlers. This is necessary because we allow the developer to assign subjects a set of permissions they are allowed to use for each channel and do not assume any cooperation other than entering policy rules. In particular, in the case of JavaScript interfaces and event handlers, we need to know beforehand what permissions the requested method uses in order to determine if the subject under investigation is granted all of the permissions required by that method. DRS employs an *Information Unit* which hosts and manages the two data structures corresponding to policies (policy map) and the permissions used by class methods and event handlers (permission map). During invocation on any of the three aforementioned channels, DRS intercepts the invocation and checks if any one of the policy rules allows the subject to execute or access the requested part of the channel, or if the user needs to be prompted for a decision. If the request is allowed according to the developer policies or by the user, DRS lets the invocation go through, otherwise it gracefully blocks the request.

Note that it could be the case that the app itself runs web code loaded from its local files. For example, the `loadUrl()` method can be invoked with "file://" which loads a html file from the local storage or with "javascript://" which invokes the inline JavasSript code with no origin. Since the subject of this execution is the app, we treat this as trusted. However, there could be cases where web code can run with undefined origin (e.g., by loading code with the `eval` JavaScript function). Our system completely blocks such attempts.

Parsing module. Draco allows developers to enter policy rules into the WebView without having to modify the source code of the Android Open Source Project (AOSP). We con-

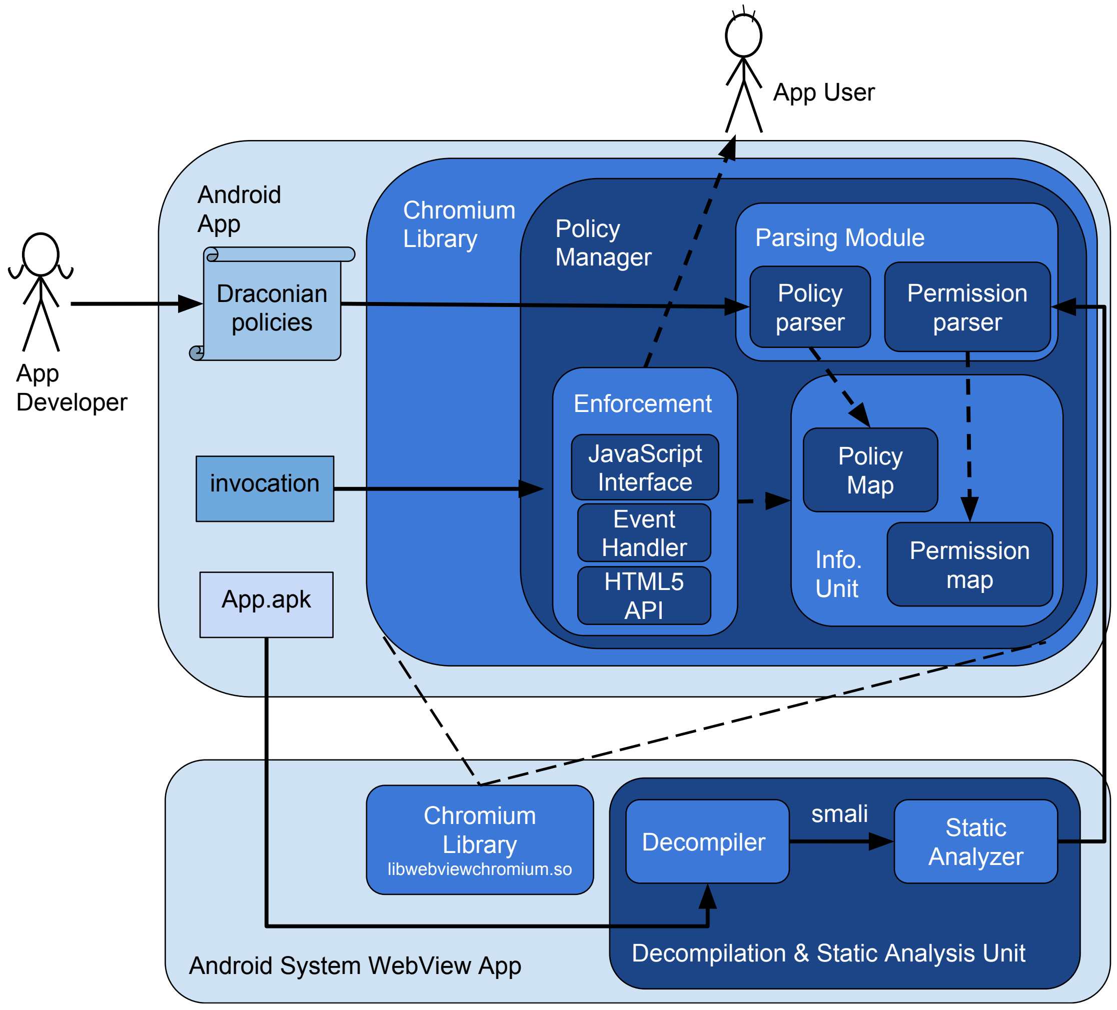


Figure 4.1: Draco Runtime System (DRS) architecture.

sidered extending the Android Manifest file with policies (as was done in previous work), as well as using annotation tags as policy rules on the exposed Java code. However, both of these approaches require changes to be made on the Android platform itself; the former requires changing the parsing logic in the *Package Manager*, and the latter requires changing the WebView APIs.

In Draco, we exploit the existing `loadUrl` method in the WebView API, and piggy-back rules into the WebView. As we explained before (see Section 2.1.5), the `loadUrl` method takes a URL string as an input argument and loads this URL. It can also execute JavaScript code if the given string starts with the *"javascript:"* tag. We extend the functionality of `loadUrl` within the WebView system app, to capture strings that start with the *"policyrule:"* tag which indicate a Draconian policy rule for the WebView. DRS's *Policy Manager* class is implemented in the facade Java layer of Chromium, and is responsible of parsing DPL rules and inserting them into the data structures we utilize for enforcement. This class uses the Java Native Interface (JNI) to talk to its native “back end” that performs these functionalities in C++. This structure makes it easier to communicate the policies to the Chromium implementation that performs most of its main functionalities in native code (C++).

App decompilation and static analysis unit. For the enforcement of permission-based DPL rules that regulate the use of sensitive APIs in the JS interface and event handler channels, we need to determine the permissions used in a given class and its methods for JavaScript interfaces, and permissions used in the event handler callback methods for the event handlers channel. Such a permission-based rule might be expressed informally as follows: “Origin X can access a method which makes sensitive API calls that require permissions Y, Z only if it’s given these permissions by the developer.”. Doing this without the help of the developer (e.g., by submitting the list of methods with the used permissions) requires static analysis on the application code to retrieve the permissions used in the JavaScript interface classes and their methods, and in the event handlers. DRS uses apktool [64], which works with a success rate of 97.6% [68], to decompile the app into smali bytecode, then finds the exposed JavaScript interface classes and the event handlers declared in DPL rules, and processes them to find the permissions that are required by them. This requires determining the sensitive API calls made in the class methods and the event handlers. DRS utilizes the API-to-permission mapping released in PScout [69] and searches for the API calls specified in this mapping in the app’s decompiled smali code. This provides DRS with a list of used sensitive API calls and their corresponding permissions required by each class method and event handler. DRS performs this process in the background, only during app installation and update time and saves the results in a file in the app’s data folder to avoid repetition of this process.

Information unit. At the core of the PolicyManager lie two components: the *policy map* and the *permission map*. The *Policy map* uses a hash map to keep track of the developer DPL rules (as key value pairs). It uses a string for its key, and a vector of strings for the value. The key consists of the subject, channel name, and type of the channel option, namely class name (for interfaces), function (for event handlers), or permission (all channels). In case the channel option is a JavaScript Interface class name, the vector of strings (value for that key) consists of the name of the allowed methods for that interface. As for permission for the channel option, this vector will be the name of the allowed permissions. The *Permission map* is used for tracking the used permissions in a given class and its methods for JS interfaces, and for tracking permissions used in event handler methods. The list of permissions for these methods will be retrieved from a permission file created by the aforementioned static analysis unit.

Enforcement. Chromium works based on a multi-process architecture (Figure 4.2). Each tab (called Renderer in Chromium jargon) in the browser is a separate process and talks to the main browser process through Chromium inter-process communication (IPC). Even though WebView is based on Chromium, it does not inherit this multi-process architecture. Instead, it keeps the same code structure but adopts a single process architecture, due to various reasons including the difficulty of creating multiple processes within an app in Android, concern for memory usage in resource-limited environments, synchronization requirements of Android Views, as well as other graphics related issues. DRS policy enforcement is implemented in the native part (C++) of the `Browser` component of the Chromium code 4.2. This is because this is the point of invocation for all of the channels and most of the necessary information to perform access control already resides in this component.

• *JavaScript interfaces.* For the JavaScript interfaces, the origin information is not passed to the Browser component with the creation of a bridge object. Hence, in the Renderer, we get the security origin (as in same origin policy) of the calling frame and propagate this information to the Browser. When the invocation happens, we know the name of the class method the origin wants to execute, however, we have no way of knowing the class name of the object this method belongs to since C++ provides no way of retrieving the name of a Java object's class in run time. That is why, the retrieval of the class name needs to be done in the Java facade layer in Chromium in order to be communicated later to the native layer. This is done after the call to `addJavaScriptInterface` (so that we do not have to change this API method) and before the object moves to the native side (so that we still have class name information). Given the channel name, class name, method name and the web origin, the *Policy Manager* is able to make an access control decision for this origin using the aforementioned data structures to check if the required policies exist for this domain to execute the method. The *Policy Manager* first checks if the origin has access to the given method of the class, and if that is the case, it performs the permissions checks to see if the permissions given to the origin would be a superset of the permissions used by the requested method. If the origin passes both checks, then it is allowed to perform the invocation of the requested method. However, if the decision point set in the respective DPL rule is set to "user", then the user is prompted with the description provided for this rule to make the access control decision.

• *Event handlers.* For the event handler channel, all the information necessary for enforcement exists in the Browser component. Hence, given the channel name, origin, and the name of the event handler method, the *Policy Manager* can enforce the policy. The enforcement

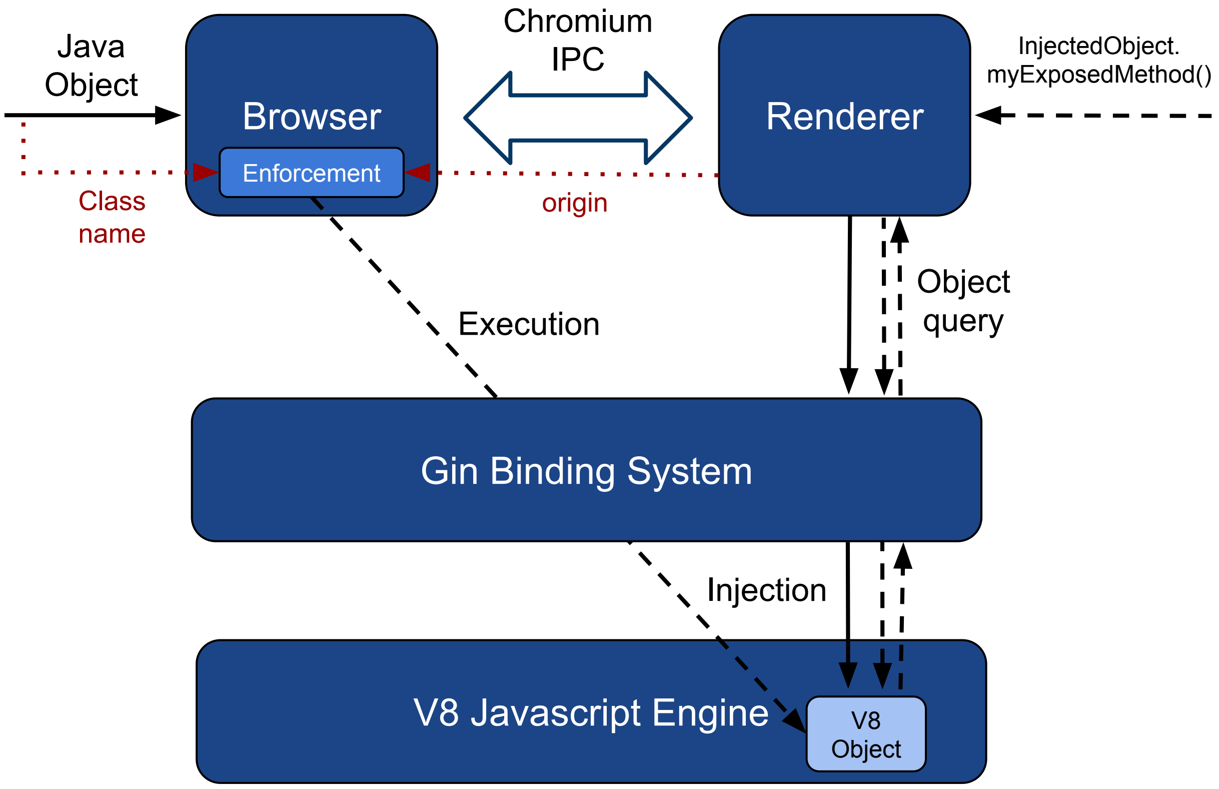


Figure 4.2: Enforcement in Draco implemented in the Browser component in Chromium for all channels. For JS interfaces, Java objects are inserted as V8 objects into Chromium. Draco needs class name and origin information from outside the context of Browser component.

logic is similar to that for JavaScript interfaces. The *Policy Manager* first checks if the origin is allowed to execute the event handler method for a given JavaScript event. If so, it then performs permission checks on the event handler to see if the origin is granted the permissions to execute the handler. If that is the case, then the origin is granted access to this event handler.

• *HTML5 API.* DRS intercepts the entry point of the `onGeolocationShowPrompt` callback for geolocation and the `onPermissionRequest` callback for other HTML5 resources in the native part of Chromium code for this channel. We retrieve the resources from the request object in Chromium and for each resource ask the Policy Manager whether there is a rule allowing the origin to use that permission, or ask the user if the developer specifies in the DPL rule for the user to be consulted. If the origin is allowed by a DPL rule (and by the user if needed), DRS allows the origin to go through with the invocation, else DRS gracefully blocks the request.

4.4 SYSTEM EVALUATION

In this section, we evaluate the effectiveness and performance of Draco on commercial off-the-shelf (COTS) devices.

4.4.1 Effectiveness

In Section 4.2, we described attacks on the CVS Caremark and Job Search apps. To evaluate the effectiveness of Draco, we enhanced the apps with DPL rules as these are described in Section 4.3.2. We have installed the apps on a Nexus 5 phone running Android 6.0. We have also updated the WebView system app to a version enhanced with the Draco Runtime System. In both cases, we found that Draco successfully blocks all illegitimate access attempts by spurious domains. At the same time, the legitimate domains can function properly and the app remains fully functional.

4.4.2 Performance

As explained before, DRS consists of: 1) a static analysis unit for determining the permissions used by class methods and event handlers, 2) a parsing module that dissects the policy rules entered by the developer and permissions used by class and event handler methods and inserts them into our data structures, and 3) an enforcement unit that intercepts invocations on JS bridges and invocations made via HTML5 APIs to ensure that the origin making the request is granted the access right by the developer and to block the request otherwise, or to prompt the user for granting permissions. We will analyze the performance of each component separately. We conduct all of our experiments on a LG Google Nexus 5 smartphone, which runs Android 6.0 (Marshmallow) and is equipped with 2.26GHz quad-core Qualcomm Snapdragon 800 processor and 2GB RAM.

App decompilation and static analysis unit. We first require the decompilation of class files associated with Draconian policy rules into smali bytecode. Currently, we are performing this off-line (not on Android); however, there are existing tools that can decompile apps on the Android platform. For example, apktool [64] decompiles all the resource and class files in the target app between approximately 60 to 90 seconds, depending on the size of the app. Another tool, DexDump [70], can perform app decompilation much more efficiently since it parses classes on demand. We hope to borrow from their techniques and use them in our future implementation.

Table 4.2: Runtime for static analysis on Nexus 5

Static Analysis cost	average (s)	standard deviation
small class (5 methods)	3.004	0.054
medium class (10 methods)	5.940	0.114
large class (15 methods)	8.766	0.167

After decompiling the target app, we statically analyze the classes associated with Draconian policy rules to determine which permissions are used by each exposed method. Table 4.2 shows the performance results of permission extraction of class methods for three cases: 1) a small class with 5 methods (506 smali instructions), 2) medium-sized class with 10 methods (1106 smali instructions), and 3) a larger class with 15 methods (1706 smali instructions). Here, the total number of smali instructions in methods is the dominating factor for the performance since for each instruction we perform a lookup in our data structure for PScout mappings.

Policy parsing module. The Parsing module runs each time the app is launched by the user in order to populate the in-memory policy and permission maps. As explained before, there are two sub-components of the parsing module; the *policy parser*, which simply parses the policy rule inserted by the developer, and the *permission parser*, which parses the output of the static analysis to identify the permissions used by class methods and event handlers for the classes and event handlers declared in the DPL rules. Both of these components incur minimal overhead: in total, we identify the run time of the parsing module to be in the order of miliseconds as shown in Table 4.3. This cost is negligible compared to the launch time of Android apps, which is expected to be in the order of seconds [71].

• *Policy parsing.* The complexity of the inserted DPL rule can change the total run time of the parsing operation. In order to investigate how rule complexity affects performance, we have considered three types of rules: 1) a *simple* DPL rule that involves 5 class methods, 2) a *large* DPL rule that involves 15 class methods, and 3) a *semantically large* DPL rule that involves all class methods (i.e., uses *all* tag). Listing 4.2 shows how these rules can be added to the app. We only investigate the performance for the JavaScript interface channel; however, the results are comparable for other channels since the construction of the policy rules are similar. We run each experiment 10 times and report average run time and standard deviation. Table 4.3 shows the results for the three types of policy rules (case 1, 2, 3). First we observe that the parsing overhead, is negligible, in the order of a few miliseconds. Additionally, as the number of class methods increases (from 5 to 15), the run time slightly increases. This is simply because number of insertion operations performed is

Listing 4.2: Small, large, semantically large rules for JS interfaces

```
// small policy rule
mWebView.loadUrl("policyrule;allowjsinterface;
https://mydomain.com;GeoWebViewActivity$JsObject<classMethod1,
classMethod2,classMethod3,classMethod4,classMethod5>");

// large policy rule
mWebView.loadUrl("policyrule;allowjsinterface;
https://mydomain.com;GeoWebViewActivity$JsObject<classMethod1,
classMethod2,..., classMethod14, classMethod15>");

// (semantically) large policy rule
mWebView.loadUrl("policyrule;allowjsinterface;
https://mydomain.com;GeoWebViewActivity$JsObject<all>")
```

proportional to the number of class methods in the rule($\mathcal{O}(n)$). However, for a policy that addresses all class methods, the run time is even smaller than that of a simple policy. This is because *all* tag semantically means that all class methods are involved, and when it is used, we do not require insertion of all the class methods into the policy map one by one.

• *Permission parsing.* The performance of permission parsing is affected by the the number of permissions used by the app, and the number of sensitive API calls (i.e., required permissions for the sensitive API call) used in each method. It has been shown that on average Android apps use five permissions [72]. Therefore, we consider two cases: 1) a small permission file (created by the static analysis unit) for a class with five methods where each method uses five or less permissions, 2) a large permission file with 20 class methods and five permissions for each method. We consider the second case to be not very likely to occur and use it only as an upper bound on the performance for permission parsing. For both of these cases, we use a simple policy rule (including 5 methods), with the addition of the permission list that contains all the five permissions the app (without using *all* tag) grants to the given subject. Table 4.3 shows the results of policy parsing (which include permission parsing). As can be observed from the table, the run times are still in the order of milliseconds, with the total run time being higher for when the permission file that contains many methods that use all of the app permissions.

Enforcement. It is important for the enforcement to be efficient since this an action that is expected to be performed frequently during the lifetime of an app. Thus, any delay can affect the app's run time performance and degrade user experience. Here, we take a closer look at the performance of the enforcement unit for the JavaScript interface and HTML5 channels.

Table 4.3: Runtime for policy rule parsing and insertion on Nexus 5

Parsing cost	average (ms)	standard deviation
(1) small policy	1.874	1.248
(2) large policy	2.453	0.811
(3) semantically large policy	1.633	0.847
(1) w/ small permission file	2.428	0.820
(1) w/ large permission file	8.434	2.269

Table 4.4: Runtime for enforcement on JavaScript interface channel on Nexus 5

Enforcement cost	average (ms)	standard deviation
small policy (allow)	0.356	0.260
small policy (block)	0.243	0.051
large policy (allow)	0.965	1.214
large policy (block)	0.551	0.124
semantically large policy (allow)	0.146	0.0252

We do not present our results for the event handler channel since they are intrinsically similar to those of the JavaScript interface channel. For the former cases, we report the average time it takes (and standard deviation) to allow and disallow an origin.

• *JavaScript interface channel.* We take the same approach as in our evaluation for policy parsing, and perform enforcement corresponding to small (5 methods), large (15 methods) and semantically large (all methods) policy rules. For each case, we assume the origin wants to access the method that is the last method in the provided method list so that we get an upper bound on the run time (since we perform linear search in vector that contains the methods associated with a policy rule). Table 4.4 shows the results for the JavaScript interface channel. Evidently, the enforcement overhead is negligible (in the order of microseconds).

• *HTML5 API channel.* For the HTML5 API channel, we consider two cases: 1) an access control decision is made solely by the system, and 2) the user is prompted to make a decision on the use of permissions. Table 4.5 shows the time taken by the system for both of these cases for the HTML5 API channel. We do not show the time the user takes to grant or revoke access to permission-protected resources. Naturally, this will be at least in the order of seconds with a large variation, and is many orders of magnitude larger than the purely-system based access control decision. Again we observe sub-millisecond delays highlighting the efficiency of DRS enforcement.

Table 4.5: Runtime for enforcement on HTML5 channel on Nexus 5

Enforcement cost	average (ms)	standard deviation
system (allow)	0.282	0.093
system (block)	0.130	0.029
user (allow)	0.326	0.116
user (block)	0.286	0.076

4.5 SUMMARY

In this chapter, we investigated the overlooked JavaScript bridge vulnerabilities for native mobile web applications that use embedded web browsers (WebView) to display web content. We showed cases where highly-popular Android apps become vulnerable and inadvertently expose their internal resources to untrusted web code. By investigating the use of WebView APIs by app developers, we identified the need for a unified and fine-grained access control mechanism on WebView. Hence, we proposed Draco, a unified access control framework that allows developers to declare access rules for the exposed resources with fine granularity and enforces these access policies at runtime. Draco's declarative policy language can be used by app developers to create policy rules that specify their trusted or semi-trusted origins with capabilities defining their access coverage on the three access channels (JavaScript inteface, event handlers, HTML5). Draco Runtime System then enforces these policy rules in an effective and efficient manner. This approach also saves developers from implementing burdensome programming measures (i.e., navigation control, multiple WebViews with different levels of exposure) in an attempt to prevent exposed resources from web domains. Draco is easily deployable since it does not require Android OS modifications, but only enhancements in the Android System WebView app.

CHAPTER 5: SECURING INTER-PROCESS COMMUNICATION

As one of its main access control mechanisms, Android utilizes a permission model in order to regulate access to sensitive user and platform resources. In this chapter, we will present our security analysis on a key part of this model, namely custom permissions, which is an access control mechanism to secure inter-process communication on the Android platform. We found custom permissions to be suffering from serious design issues that enable malicious apps to escalate their privileges to obtain unfettered access to critical device resources as well as permission-protected components of other apps. We will present our systematic and formally-verified re-design of custom permissions to eradicate these perennial issues. This chapter is based on our work in [6].

5.1 INTRODUCTION

Android's permission model forms the security basis for the critical operations that can be performed on the platform by the apps. In a nutshell, the main purpose of this model is to regulate access to platform and app resources, which is achieved by utilizing a set of security labels, called permissions. In order to protect the platform resources (e.g., microphone, Internet etc.), the platform uses *system permissions*, which are a predefined set of permissions introduced by the platform itself. The permission model also provides the platform with finer-grained security as a means to protect Inter-Process Communication (IPC) between different app or system components. Specifically for this purpose, Android introduces *custom permissions*: these are application-defined permissions which allow developers to regulate access to their app components by other apps. In fact, the use of custom permissions is very common among third-party applications. According to our study on the top free apps on the Google Play Store, 65% of the apps define custom permissions, while 70% request them for their operation.

Unfortunately, design flaws and vulnerabilities in custom permissions can completely compromise the security of IPC, inevitably leading to exploits on third-party apps and the platform itself. Previous work has consistently found custom permissions to be problematic [50, 51], and as a response to these studies, Google made an effort to address the identified problems by releasing bug fixes. However, similar vulnerabilities still exist even after Google patched this initial wave of vulnerabilities. In this work, we present two classes of attacks that exploit the vulnerabilities in custom permissions to get unauthorized access to platform and app resources. With one of our attacks, a malicious app can bypass the user interaction requirements for acquiring dangerous system permissions on Android versions that support runtime permissions and stealthily access high-risk platform resources (e.g., camera, microphone etc.). With our other attack, a malicious app can escalate its privileges to gain elevated access to the protected components of other apps. We further demonstrate how an adversary can utilize the aforementioned vulnerabilities to target high profile apps with millions of downloads, such as *CareZone* and *Skype*, and access sensitive user data (e.g., medical conditions, insurance information) and functionalities (e.g., VoIP calls). We have officially reported these attacks to Google, which acknowledged them as severe flaws that need to be addressed in the next versions of Android.

In our investigation of the Android permission model and its respective source code, we observed that *there is no separation of trust between system and custom permissions in the Android framework*, which leads to the manifestation of permission vulnerabilities; we call this failure to distinguish custom permissions in the system, the 'predicament' of custom permissions. First, system and custom permissions are currently insufficiently isolated and they receive the same kind of treatment from Android, which opens up opportunities for malicious apps to utilize custom permissions to obtain unauthorized access to platform resources. Second, there is currently no enforced naming convention for when declaring custom permissions—apps are allowed to declare custom permissions with *any* name they desire. This creates a confused deputy problem where a privileged app's protected resources can be utilized by unauthorized apps that possess different custom permissions declared with the same name as of the ones used by the privileged app to protect its resources. In order to systematically address these problems, we propose a design and corresponding implementation which we call *Cusper*. Cusper decouples the handling of custom permissions from system permissions to prevent an adversary from escalating their privileges and stealthily acquiring system resources. Additionally, Cusper implements an OS-level naming convention for custom permissions to prevent custom permission spoofing. This is backward-compatible with existing apps and enables the system to properly identify custom permissions according to the developer signature of their definer apps.

To prove the correctness of Cusper, one could employ traditional analysis methods such as testing and static analysis. However, these are typically insufficient since they depend on the analyst determining all possible test cases, a challenging endeavor. In contrast, formal methods can be leveraged to build models and verify that key properties are never violated in a proposed system. In fact, previous work has already used the latter approach to formally model the Android permission model [42, 43, 44]. Unfortunately, the proposed formal models are outdated since they correspond to the *install time* permission model of Android. In order to verify the correctness of Cusper, we build the first formal model of the Android *runtime* permission model using the Alloy specification language. Specifically, we model the data abstractions for permission-related structures and the behavior of system operations (e.g., install, uninstall, update) that concern permissions for the runtime model, which significantly differ in terms of both abstractions and behavior from the previous model. We found that the original permission model violates two fundamental security properties regarding access to app and platform resources: 1) there should be no unauthorized component access, and 2) there should be no access to high-risk ('dangerous') platform resources without user's consent. We leverage our formal model to demonstrate the existence of vulnerabilities that violate these invariants in the original permissions model and show that, in contrast, Cusper *always* satisfies them. Finally, to illustrate Cusper's practicality, we implement it in Android and show that it effectively resolves the identified vulnerabilities while incurring a negligible overhead.

5.2 USE OF CUSTOM PERMISSIONS

Custom permissions provide security to IPC that apps harness for their operation. They are utilized by app developers to restrict access to components as per the sensitivity of the protected resource. In this section, we investigate the prevalence of custom permissions among the top free apps on Google Play and showcase two high-profile apps that we selected to launch attacks on by exploiting the vulnerabilities in custom permissions.

5.2.1 Prevalence

We collected 50 top free apps from each of the Google Play Store categories (with some failures in collection) and in total analyzed 1308 apps to identify statistics regarding the use of custom permissions. As can be seen in Table 5.1, 65% of the apps in our dataset declare custom permissions (statically or dynamically) and 70% of them request custom permissions. Additionally, 89% of all the permissions created by these apps are of protection level signature (see Table 5.2), which indicates that app developers typically use custom permissions to allow other apps to utilize their protected components only if they are signed by the same developer or company.

This analysis shows that custom permissions are commonly utilized by app developers. Vulnerabilities in their use create risks for the security of the platform and key apps. To illustrate this we have identified ways we can launch attacks on the platform to obtain any system resource (e.g., camera, microphone) without user consent and on apps to stealthily access their protected components and data. Apps that utilize custom permissions are

Table 5.1: Apps at risk due to custom permissions

Usage	Number of Apps
Create Static Custom Permissions	834 (64%)
Create Dynamic Custom Permissions	50 (3%)
Create Custom Permissions	847 (65%)
Request Third-party Permissions	919 (70%)
Total number of apps in dataset	1308 (100%)

Table 5.2: Protection Levels of Custom Permissions

Permission Protection Level	Number of Permissions
Signature Permissions	1203 (89%)
Dangerous Permissions	14 (1%)
Normal Permissions	40 (2%)
Signature or System Permissions	57 (4%)
Total Number of Permissions	1350 (100%)

potentially susceptible to the attacks that target their protected components when they are installed on Android 6.0 or newer. This currently includes more than 50% of all Android devices [73] with a growing user base. This is a widespread security concern: just the apps in our dataset have been downloaded on average 25 million times.

5.2.2 Case Studies

In this section, we present case studies where sensitive data or resources of popular Android apps have been leaked through the custom permission vulnerabilities. In order to obtain the vulnerable components that leak resources, we conduct manual analysis which requires going through the decompiled application code and crafting attacks specifically for the app in study. As an example, to attack a vulnerable activity component, we go through the decompiled code to find the Java file for the activity itself and if it is not obfuscated, we proceed to inspect the source code to identify whether the intent it expects is of a particular format. Finally, using this information, we create attack apps that exploit the existing custom permission vulnerabilities and stealthily activate the component of interest with the appropriate intent. We will explain the details of how this attack works in Section 5.3.2.

CareZone is a medical Android app produced by a company with the same name. It has 1,000,000+ downloads and has a 4+ rating on the Google Play Store. The app allows users to store medical-related information such as health background (e.g., blood type, medical conditions, allergies etc.), medication lists, medical contacts and their addresses, calendar events, insurance information, and photos or health files in an organized manner. It also features calendars to track appointments as well as to-do and notification lists for tracking tasks. All of this medical data and meta-data is stored in a single content provider which has been exported and is protected by a signature permission. There are no other dynamic checks on accessing the content provider. Our attack is able to bypass the signature requirements and read the entire content provider, which gives us access to the aforementioned sensitive data without the user's explicit consent or knowledge. The fact that all this data was stored in a single content provider seems to reflect the developer's implicit reliance on the security guarantees provided by the platform.

Skype is an Android app by Microsoft that allows users to make voice and video calls over the Internet. It has 500,000,000+ downloads and a 4+ rating. Skype has an activity which can be invoked to start the call functionality to any telephone number. This activity is protected using a signature permission which, once bypassed, would allow the adversary to invoke calls to a specified person or number. This could have many implications. For example, it could be used as part of a suite of other spying capabilities. Our attack is able to spoof the original permission and launch Skype calls without the user's knowledge through this protected activity.

5.3 ATTACKS

Custom permissions play an important role in enabling re-usability on the Android platform by providing security to IPC; hence, any threat to their proper operation can result in the compromise of the security of the apps and the platform itself. In this section, we discuss two types of custom permission vulnerabilities we identified on Android: 1) custom permission upgrade and 2) confused deputy. By exploiting these vulnerabilities, an app can bypass user consent screens for granting/denying permissions to obtain high-risk system resources and can also gain unauthorized access to protected components of other apps. We reported these to Google which acknowledged both of them as severe vulnerabilities. Given its real-world implications and its prevalence, we believe that the predicament of custom permissions constitutes a current and notable security risk worth addressing.

Threat Model. We consider an adversary that has the ability to crawl app markets (e.g., Google Play Store) to download victim apps of interest, reverse engineer them by utilizing several tools [64, 63, 62], and analyze the Android manifest files and source code of these apps to observe the cases where custom permissions are used to protect app components.

The adversary can build and distribute on app markets a set of malicious apps that exploit the custom permission vulnerabilities of Android to launch attacks on the victim apps and on the platform.

5.3.1 Custom Permission Upgrade Attack

Android runtime permission model (supported by Android 6.0 and onward) requires user's approval for granting apps permissions of protection level `dangerous`. This attack enables a malicious app to completely bypass the user consent screen and automatically obtain *any* `dangerous` system permissions [74].

In particular, there are 24 dangerous permissions in 9 permission groups [75] on the current version of the Android platform (7.0), which protect access to high-risk system resources (e.g., storage, contacts, location, camera, microphone, sms, sensors etc.). Our attack illustrates how an adversary can gain unfettered access to *all* high-risk system resources that are protected by these permissions without the user's consent. Google acknowledged this attack as a serious security vulnerability (CVE-2019-2200) [5].

Attack Overview. First, the adversary creates an app that includes in its manifest file a custom permission declaration with the protection level `normal` or `signature` and sets this custom permission to be a part of a system permission group (e.g., storage, camera etc.). Then, they update the definition of this custom permission so that the protection level is changed to `dangerous` and proceed to push an update to their app on the respective app market. Here, this update can be pushed to all the app users after the app reaches a target user base. In addition, specific user groups can be targeted via the use of push services (e.g., Google Cloud Messaging (GCM) [76], which is used by 94% of the apps in our database that utilize custom permissions) that allow sending update notifications, and via enterprise app stores (e.g., Appaloosa [77]) that enable enforced targeted updates. The expectation is that since the custom permission is of level `dangerous`, the user will be prompted at runtime to make a decision on whether to grant or deny this permission in the runtime permission model. However, the malicious app automatically gets granted the permission. In addition, since the runtime permission model grants dangerous permissions on a group basis, the app also automatically obtains all the other requested dangerous permissions of the system permission group that the original permission belongs to. Same procedure can be followed to attack *any* system permissions group; hence, the adversary can silently obtain *all* system permissions simultaneously. Requesting dangerous permissions in the Android manifest constitutes no problems for the adversary, as permission requirements of an app are not directly presented to users at installation since Android 6.0. Hence, the user will be completely unaware that all these system permissions are granted to the app.

Internals of the Attack. Android does not treat custom permissions any differently than system permissions. As we will describe in more detail in Section 5.5.2, granting of any permission is handled according to the permission's protection level and the SDK level of the requesting app on the runtime model. Normal and signature permissions are always granted as install time permissions. For legacy apps (SDK level <23), dangerous permissions are still install time permissions; whereas for new apps they are granted at runtime. In case a legacy app gets *upgraded* to SDK level 23 or more, the system also "upgrades" the granted dangerous permissions from install time to runtime permissions and automatically grants them if they were not manually revoked by the user through the permission settings (indicated by the `FLAG_PERMISSION_REVOKE_ON_UPGRADE` flag being set for the permission). However, the system does not consider other cases where a change in the definition of a permission can mistakenly trigger the same *permission upgrade* operation. In the case of our attack, when a custom permission declaration is modified by an app update such that the protection level changes from normal or signature to dangerous, the system wrongfully treats this case as *an app upgrade*, and tries to also upgrade the existing permission to a runtime permission even though the update operation did not change the app's SDK level. Since Android does not allow users to revoke normal or signature permissions, the aforementioned flag will never be set for the existing install permission. Hence, the dangerous permission will be granted automatically without any user consent. Evidently, this violates a key security principle that should always hold in the Android runtime permission model: *no dangerous runtime permission should be granted without user interaction.*

Note that, the problem here is that the system does not consider the special cases that can happen in the case of custom permissions. If a *system* permission is being upgraded from an install to a runtime permission for an app, this can only mean that the app is being upgraded to SDK level 23 or more. However, when a *custom* permission is upgraded for an app, this can indicate either that the legacy app is being upgraded, or that the permission definition is being changed from normal or signature to dangerous. Currently, the system is not equipped with the ability to distinguish between these two cases for custom permissions as it cannot even distinguish system permissions from custom permissions. To make things worse, the system allows a third party developer to declare a custom permission as a part of a system permission group. Thus, the adversary can not only get a dangerous custom permission silently granted, but they can further get access to all system dangerous permissions in the same group with any granted dangerous custom permission.

5.3.2 Confused Deputy Attack

In this attack, the adversary exploits the lack of naming conventions for custom permissions on Android to launch an attack on a victim app that utilizes custom permissions to protect its components [78]. To do this, the adversary counterfeits the custom permissions of the victim app by reusing their names in her own permission declarations and takes advantage of the system's inability to track the true origin of permissions to access protected components of the victim app. Google acknowledged this attack as a high-severity security vulnerability since it bypasses operating system protections that isolate application data from other applications (CVE-2017-0593) [4].

Attack Overview. In this attack, the adversary's goal is to get the operating system to grant their apps a signature custom permission of a victim app that is signed by a different key than that of the adversary and therefore obtain unauthorized access to the components protected by this signature permission.

In order to achieve this, the adversary develops two applications: 1) a definer attack app which spoofs the custom permission of the the victim app by reusing the same permission name but changing the protection level to `dangerous`, 2) a user attack app which only requests this permission in its manifest file. The reason adversary needs two apps to carry out this particular attack is that Android currently does not allow two applications that declare a custom permission with the same name to coexist on the same device. Hence, the adversary's app cannot simultaneously exist on the device along with the victim app if it declares a permission with the same name to the one used by the victim. However, the adversary can divide their attack into two different apps, one that spoofs the custom permission as long as the victim app is not installed on the device, and a second one that only requests this permission and is able to coexist with the victim. The definer attack app needs to be installed first by the user, and this should be followed by the installation of the user attack app. After the spoofed permission of the definer attack app is granted to the user attack app at runtime, the definer attack app can be uninstalled by the user or updated by the app developer (for all users or targeted to a specific group by using services like GCM or Appaloosa) to remove the custom permission definition so that the victim can be installed afterwards. After the installation of the victim app, the user attack app is able to launch an attack on the victim to freely access victim's signature-protected components even though it is not signed with the same app certificate as the victim.

Note that there can be many ways for an adversary to get the user to install two applications on their device. For instance, the app developer can use in-app advertisements and links to direct the user to app stores to download their other app (e.g., Facebook and

Messenger). Another effective way would be to utilize a common Android app development practice called plug-in architectures [79, 80], which on demand unravel new features to the user in the form of new apps in order to foster re-usability and save storage space by unlocking features only if they are necessary. An example to apps using this architectures is Yoga Guru [81], which unlocks users new yoga exercises—as part of new apps—only after making progress with the set of exercises they currently have.

Internals of the Attack. During the installation or the update of an app, if a permission definition is removed from the system due to this operation, the system iterates over the existing apps to readjust their granted permissions. The desired outcome of this behavior is that all the undefined permissions should be revoked to the remaining apps after an uninstallation or an update. However, instead of immediately revoking an undefined permission to the remaining apps, the system instead revokes it only if a new permission with the same name is being redeclared. Even though this initially seems unharmful since the granted permission is rendered useless until it is redefined, this behavior is what enables our attack. Once the permission name is recycled by the introduction of a new signature permission, due to the mismatch of signatures, the system attempts to revoke this permission to the adversary; however, it mistakenly only revokes install permissions and fails to do so for runtime permissions. This, in turn, leaves the undefined runtime permissions granted to the app. Since Android utilizes only the names of permissions during permission enforcement, it cannot differentiate between two distinct permissions with the same declared name. Hence, the app holding a "dormant" dangerous permission gains unauthorized access to components protected with a signature permission with the same name. Evidently, this violates a key security principle that should always hold in the Android permission model: *there should be no unauthorized component access.*

Note that again the framework developers seem to disregard the peculiarities and corner cases created by custom permissions. Currently, a third-party app cannot define a custom permission using the name of an existing system permission. It was, however, possible for them to use the name of a new system permission that were to be defined in the next version of the OS, to hijack the system permissions [82]. Google's fix to this security vulnerability was that the system would always take the ownership of permissions defined by itself; hence, spoofing attacks on system permissions should not succeed anymore as the platform is treated as the main principal to define/remove system permissions under any circumstance. However, a similar approach cannot be applied to custom permissions as the system cannot make a decision regarding the ownership of a permission between two apps that define the same custom permission. Hence, we not only need to identify whether a permission is system or custom, but in the latter case, we also need a way to identify its origin and treat it as a different permission in case there are other permissions with the same name. It is worth noting that whether a permission is custom or system cannot be determined solely based on its name as even custom permissions can currently use system prefixes (e.g. `android.permission`) and system apps can create permissions with any name without being forced to use a system prefix (e.g., browser permissions).

5.4 CUSPER

System permissions are defined by the platform—a privileged principal—whereas custom permissions are defined by apps—less privileged principals. The former kind typically protects system resources while the latter is utilized to protect inter-component communication between apps. The fact that the system treats them the same, results in severe security vulnerabilities as the ones we discovered (Section 5.3). Note that other vulnerabilities might also exist or might manifest in the future because of this non-separation between the two classes of permissions. Ideally, we need a new design which will allow us to achieve a clean separation of trust between the system and custom permissions. This way, the system will have to handle the two cases differently avoiding logic errors and at the same time, any potential vulnerabilities in third party app custom permissions will not allow privilege escalation, which can enable exploits of system permissions and platform resources. However, such a new design needs to be carefully constructed to be practical. In fact, it needs to be as simple as possible to be adopted in practice, and backward compatible. A complete redesign of the Android permission model would require non-trivial modifications to the Android framework while thousands of apps relying on custom permission would be immediately affected. Instead, in our work, we introduce two main design principles which can easily be incorporated into the current design of Android permissions, require no changes to the existing apps, and can guarantee a separation of trust eliminating the threat of privilege escalation in permissions, without breaking the operation of system and third-party components that rely on permissions. These design principles are: (a) decoupling of system and custom permissions; (b) new naming scheme for custom permissions. We implement these in our system that we call Cusper.

5.4.1 Isolating System from Custom Permissions

Currently, Android does not maintain distinct representations for system and custom permissions, that is, the system does not track whether a permission originated from the system or from a third-party app. Due to this reason, both types of permissions are also granted and enforced in the same fashion. As we have shown in Section 5.3, this is problematic as it allows apps to use custom permissions to gain unauthorized access to system permissions. For example, a malicious app can declare a custom permission and assign it to a system permission group. This behavior is allowed by Android since it does not differentiate between the two permission types. Thus, when the custom permission is granted, the app automatically gains access to the system permissions in the same group, essentially elevating its privileges from a permission defined by a low trust principal to permissions defined by the platform. In our system, we *never allow custom permissions to share groups with system permissions*. Additionally, the fact that Android internally treats all permissions the same way is an important limitation with security repercussions: platform developers tend to overlook the existence of custom permissions when handling permissions. The *custom permission upgrade attack* is an example of that. To overcome this, in our system, *system and custom permissions have distinct representations in the platform*. By doing this, we can differentiate between the two types of permissions during granting as well as enforcement and apply different strategies depending on the type of permissions.

Implementation. In order to decouple the two permission kinds, one could create separate object representations and data structures. This would require a complete redesign of the Android permission implementation throughout the Android framework which we think is impractical. Alternatively one could use existing fields in the current permission representation in Android which can give us information on the source of a permission. `BasePermission` class has a `sourcePackage` field that indicates the originating package of a permission. For system permissions defined in the platform manifest, this field is set to `android`, for system permissions defined in system packages, it *usually* starts with `com.android`, and for custom permissions it is the package name of the defining third-party app. However, the package name itself cannot be used to identify whether a package is system or third-party, as there are already system apps with package names not starting with `com.android` (e.g., browser) and even third-party apps can have package names starting with the system prefixes (`com.android` etc.). Hence, `sourcePackage` is not a reliable identifier of whether a permission is custom or system.

Instead, a both practical and robust approach, would be to extend the object representation of a permission with an additional member variable, indicating whether this permission is a custom permission. In Cusper, we implement this by augmenting the `BasePermission` and the `PackageParser.Permission` classes. The value of the new variable is assigned when an app's manifest is parsed (`PackageParser.java`) during installation or upgrade. If the app under investigation is untrusted (as indicated by its non-platform signature), we mark its permissions as custom. When parsing an untrusted app's manifest, we further check whether the app developer assigned a custom permission to a system permission group. In this case, we ignore the assignment, which results in the permission having no group. Moreover, if the app declares a custom permission group, we ensure it does not use a system permission group prefix (`android.permission-group`). In essence, we thwart the vulnerability while ensuring that even if future vulnerabilities manifest, there will be no escalation to system permissions.

After doing this, we can now track the creation of custom permissions by third-party apps. In order to particularly thwart the *Custom Permission Upgrade*, when a custom permission—which we can now effectively and efficiently differentiate from system permissions—is created with the protection level normal or signature (i.e., install permission), we simply set the `FLAG_PERMISSION_REVOKE_ON_UPGRADE` flag so that the permission will not be granted automatically if it is later updated to be a dangerous (runtime) permission.

5.4.2 Naming Conventions for Custom Permissions

Android allows third-party apps from different developers to declare permissions with the same name. The current solution is to never allow two permission declarations with the same name to exist on the device. While this sounds effective, it is unfortunately unable to stop the second attack we demonstrated: a definer app A declares a permission and another app B gets the permission granted. When the first app A is uninstalled and a victim app C comes in declaring and using the same permission to protect its components, it is vulnerable to confused deputy attacks from app B. We solve this problem by introducing an *internal* naming convention: we enforce that *all custom permission names are internally prefixed with the source id of the app that declares it.* Note that we do not expect app developers to change their practices. Custom permissions are still declared with their original names in the manifest files of apps to allow backward compatibility. However, in our system, the custom permission names are *translated* to `source_id` : `permission_name`. Thus, even if permission revocation such as in the above attack scenario fails, the attack will be rendered ineffective. This is because, as far as our system is concerned, the granted permission to app B will be an entirely different permission than the one app C uses to protect its components.

Choosing the appropriate source id is not straightforward. Consider for example using an app's package name as the `source_id`. This introduces two main problems. First, repackaged apps distributed on third-party application markets could use the package name of an app distributed on Google Play. Thus, the repackaged app could take the role of the *definer attack app* (see Section 5.3) and instigate a confused deputy attack. This is possible since the repackaged app and the victim app share the same package name and a permission created by the repackaged app cannot be distinguished from the one created by the victim if they share the same permission name. Second, using the package name as the `source_id` might break the utility of `signature` custom permissions for some use cases. For example, developers that have a set of applications which utilize each other's components, commonly use signature permissions to protect the components of their apps from others. Since the installation order cannot be determined in advance, each app in the set has to declare the same permission (i.e., same name and protection level) in their manifest to make sure this permission will be created in the system. If permissions are prefixed with their declarer app's package name, then the system will treat them as different permissions. Therefore, any attempted interaction will be wrongfully blocked.

In Cusper, we instead use the app's signature as the source id to prefix permission names. In the case of a repackaged app, assuming the malicious developer does not possess the private keys of the victim app developer, the declared permission will be a different permission in the system than the victim's declared permission. Moreover, utility is preserved since custom permissions with the signature level will be treated as the same permission as long as they come from the same developer, which is exactly the purpose. Note that the same scheme can also be utilized for permission tree names.

Lastly, the official suggestion to Android app developers which declare custom permissions, is to use names that follow the reverse domain name paradigm (similar to the one for package names). However, Android does not enforce this naming convention. Even though it will ignore a permission declaration with the exact same name as an existing permission, it allows third-party apps to use a system permission name prefix (e.g., `android.permission`) in their custom permission declarations. Since permission names and groups are currently the only information the system has regarding the intention and source of the permission, this treatment is at the very least hazardous. In Cusper, we address this naturally as we add prefixes to permission names and *never allow a custom permission to use a name prefix reserved for system permissions.* Since we decouple the two types, we can now identify the type and origin of permissions, and readily enforce this rule. To maintain backward compatibility and ensure that the custom and system permission names are distinct, we also ignore system permission names for custom permissions (as the original system currently does).

Implementation. To thwart custom permission spoofing attacks of any sort (including our *Confused Deputy* attack), apart from distinguishing between custom and system permissions, we further need a way to track the origin of custom permissions and uniquely identify them in the system. Towards this end, we implement a naming convention for custom permissions in Cusper. Our implementation consists primarily of a permission name translation operation to prefix the permission names with their source id to ensure uniqueness in the system. This translation happens during installation and update for the names of the declared custom permissions and requested install time permissions, and at runtime for dangerous permissions and the permissions used to protect components (guards).

At the time of installation, we allow the system to parse declared custom permission names from an untrusted app's manifest; however, we translate their names to be prefixed with the hash of their app's signature before the actual permission is created in the system. In the case an app is signed with multiple keys, we sort the hashes of the keys and concatenate them. Note that one could attempt to perform the translation in place. For example, it could perform the translation while parsing a permission name from the manifest. However, at that point, the app's certificates are not yet collected. Doing so would incur non-negligible overhead since it involves a number of file opening and reading operations (`PackageParser.collectCertificates()`). Instead, we keep the parsed data unaltered until after the certificate collection normally happens. Then, we scan the package's meta-data to perform the necessary translations. Our approach resulted in great performance savings which keep Cusper's performance comparable to the original system (see Section 5.6).

Similarly, we first proceed to translate the names of the requested permissions during installation or update. This is done to correctly grant install time permissions (i.e., normal and signature). Note that a requested permission might not necessarily exist in the system at this time and therefore the permission name translation cannot happen. For example, an app that declares the permission might be installed at a later point in time. Since the declared permission will be translated, it will essentially be treated as a different permission than the one requested, violating application developers' expectations. This is not a problem with install time permissions: the permission correctly will not be granted as its definition does not exist on the system at the time of installation, which is on a par with the behavior of the original Android OS. In the case of dangerous permissions which are granted by the user at runtime, we need to dynamically check for existing declared permissions. Therefore, we perform a requested permission translation at runtime. In particular, when a dangerous permission is to be displayed to the user, we perform a scan on all declared permissions to find a custom permission with the same suffix as the requested permission. In our implementation, we do not allow declaration of custom permissions with the same name which ensures that the scan will result in only one possible permission. This is also the current design of Android which does not allow two apps to declare the same permission. Note, however, that since we prefix custom permissions, one could extend our system to allow multiple apps to use the same custom permission names. In case of an app requesting that permission, we could readily resolve the conflict if one of the declarers has the same signature. If all declarer apps come from different developers, a mechanism similar to Intent filters could be utilized to allow the user to select the appropriate declarer app.

It is worth noting that one could alternatively create a separate hash map for custom permissions (e.g., key-value pairs of (suffix, prefix)) to avoid the linear scan for suffix lookup. However, this hash map would need to be kept consistent with the original hash map for all declared permissions in the system (e.g., tracking addition/removal of permissions), which is hard to achieve since there are multiple places throughout the Android source code where this in-memory data structure is updated or sometimes even constructed from scratch from files in persistent storage. Hence, for the sake of consistency and not breaking utility, we prefer the linear scan method and do not change the structure of the in-memory data types for permissions. As we will show in our evaluation in Section 5.6, this method does not result in any significant overhead.

Finally, as for permissions that are used to protect app components (guards), their name translation takes place at runtime during enforcement since a guard might not necessarily exist in the system at the time of installation.

5.5 ANDROID PERMISSIONS ALLOY MODEL

As a part of the software development process, to verify that a piece of software meets the requirements, it is common practice in industry to rely *only* on software testing and not provide formal proofs of program correctness for the underlying model as formal verification is highly time consuming, difficult and expensive. However, we argue that fundamental components like a permission system are naturally worth more effort as any failure in such components can make way for critical security vulnerabilities or even render the security of the whole system ineffective. Additionally, numerous security bug reports on similar issues present further proof that the current testing methodologies for Android permissions are not completely effective and a better way of proving program correctness is necessary. Hence, in this section, we focus on providing a formal model of Android (runtime) permissions and a formal proof for the correctness of our design for Cusper.

Formal verification allows us to systematically reason about our design of Cusper by covering many cases that would otherwise be difficult to investigate with static analysis or testing. This is not to say software testing is unnecessary when a formal correctness proof is provided. In fact, we still need software testing to verify that our implementation conforms to our proposed model (which is formally verified to be correct). On the other hand, "formal verification reduces the problem of confidence in program correctness to the problem of confidence in specification correctness" [83]. In other words, verification is performed not on the actual implementation but on a representation that is as close to the original implementation as possible. This is because it is challenging to perform formal verification at a scale required by source code, especially at the huge scale of the Android source code. Progress in the area does exist towards this for other programming languages [84], but such approaches are typically employed at the time of development, where the developer is required to annotate the code. This would be infeasible in our case where a large portion of the Android source code is already written. Additionally, correctness is proved *only* with respect to a set of fundamental properties that were defined based on the specification. There is *no* guarantee the system will behave correctly under any condition that was not a part of the defined properties or in case of redesigns of the system that might invalidate the model assumptions. Hence, the state of the art formal verification is not a silver bullet but still a best effort technique for proving correctness.

To analyze the security of Android permissions, previous work proposed formal models that correspond to the older Android versions which supported only install-time permissions [42, 43, 44]. Unfortunately, no such model exists for Android's currently-adopted runtime permissions. Hence, we build the first formal model of the Android runtime permissions and use it to verify the correctness of Cusper. This allows us to investigate Cusper under many cases such as all possible installation orders and app declarations. Note that having such a formal model has other benefits; for example, security researchers can use it to verify other properties of their interest on the runtime permission model. We based our model on the Alloy implementation of [42] as Alloy is a high-level specification language that is easy to interpret. However, we spend a significant amount of effort to extend this model to conform to the official specification for the new runtime permissions [18]. We analyze the security of the model through an automated analysis and show that when it is augmented with the design of Cusper, the fundamental security properties that were originally violated are satisfied. Our main contributions to the existing formal analysis on Android permissions can be summarized as follows:

- We updated the definitions of permission-related data abstractions in the model to comply with the new definitions of the runtime permission model.
- We significantly updated the permission granting scheme to comply with the complex granting scheme of the runtime model specification (e.g., permissions can be granted as either install or runtime).

- We implemented permission groups and permission granting on a group basis for dangerous permissions according to the runtime permission model.
- We enabled apps to dynamically change their manifest declarations and introduced an app update mechanism (apps could not be updated in the previous model).
- We identified and fixed the bugs in the existing model (e.g., missing signature checks for permissions).
- We demonstrated the existence of the aforementioned custom permission vulnerabilities in the model.
- We implemented our defense, Cusper, in the model to thwart these vulnerabilities and showed that Cusper satisfies the fundamental security properties.

We only model the parts of Android that concern permissions (e.g., permission-related data abstractions and operations) as it would be infeasible to model all of Android due to its large scope and complexity. Additionally, due to space limitations, we will be only be presenting the parts of our model that are key to understanding the general operation of the model or that significantly differ from the previous model. As for the actual Alloy implementation, we will present only a small part of it in this section. but the full implementation can be found in [85]. Our model can be dissected into three main parts: 1) abstractions related to permissions, device architecture, and applications on Android, 2) system operations that concern permissions, and 3) fundamental security properties to verify the correct behavior of the model.

5.5.1 Abstractions

In this section, we present the abstractions in our model that correspond to the representations of permissions, applications and devices on Android.

Permissions. Our `Permission` abstraction is on a par with what we described in Section 2.2.3: each permission is associated with a name, a source package to indicate the defining package name, a protection level, and at most one permission group. Listing 5.1 presents the Alloy implementation for permissions, protection levels and permission group abstractions.

Applications and Components. Each `Application` on Android has a unique package name, a signature used by the developer to sign the app, and a target SDK level. Additionally, each app can comprise of several components, defined by the `set Component`, where a component can be one of the four Android components. Each component can be protected

Listing 5.1: Permissions and permission groups in the model

```
sig Permission {
   name: PermName,
   protectionLevel: ProtectionLevel,
   sourcePackage :PackageName,
   permGroup: lone PermGroupName// if perm belongs to group
}

abstract sig ProtectionLevel {}
one sig Normal, Dangerous, Signature extends ProtectionLevel {}

sig PermissionGroup {
   name: PermGroupName,
   perms: Permission -> Time // set of changing perms
}
```

with a permission that we call `guard`. Furthermore, an application itself can have a `guard` to protect *all* of its components. Component guard takes precedence over the application guard in case they both exist. Each application can define a set of custom permissions and request a set of permissions.

In order to keep track of the permissions that are granted to apps, each app is associated with a `permissionsState` field that consists of a set of `PermissionData` objects which carry system flags and state information (e.g., whether a permission is granted as runtime or install time) regarding each permission granted to the app at any time. This concept of "stateful" permissions is one of the major representation differences between the runtime and the install time models.

In order to implement an app update mechanism, we need to allow apps to dynamically change the declarations in their manifest file. To achieve this, we associate the fields that require to be mutable with an object from the totally-ordered set of `Time` in order to allow pairing of the fields with different values at different time steps. Obviously, package name and the signature should be immutable since these are the unique identifiers for apps and developers. The ability to dynamically change declarations is another important feature we introduce in our model, as this gives us the ability to update Android apps that are already installed on a given device. Listing 5.2 demonstrates the application-related abstractions in Alloy.

Device. Each `Device` comes with a set of built-in system permissions and a set of custom permissions defined by third-party apps. We also include a platform signature in our device representation to correctly perform signature checks when granting signature permissions

Listing 5.2: Applications and components in the model

```
sig Application {
  packageName : PackageName,
  signature : AppSignature,
  declaredPerms: Permission -> Time, // custom permissions
  usesPerms: PermName -> Time, // requested permissions
  guard : lone PermName, // protects all components
  components: set Component,
  targetSDK: Int -> Time,
  // carries info regarding granted perms
  permissionsState: PermissionData -> Time
}
abstract sig Component { // def. shortened for brevity
  app: Application,
  guard: lone PermName, // protects only this component
}
sig PermissionData {
  perm: Permission,
  flags: Flags,
  isRuntime: Bool // runtime or install permission
}
```

defined by the system. Listing 5.3 illustrates the device abstraction in Alloy.

5.5.2 System Behavior

In this section, we describe the main system operations (i.e., Alloy predicates) that deal with Android permissions. By carefully investigating the Android source code, we have observed that most of these critical operations have either undergone a significant amount of change or been recently introduced with the runtime permission model. Specifically, apart from the significant change in abstractions, main operations such as install, uninstall and

Listing 5.3: An Android Device in the model

```
one sig Device {
  apps: Application -> Time,
  builtinPerms: set Permission, // system permissions
  customPerms: Permission -> Time, // custom permissions
  platformPackageName: one PackageName,
  platformSignature: AppSignature,
  builtinPermGroups: set PermissionGroup // system groups
}
```

Table 5.3: Cases of grantPermissions. (* Precondition: Permission should exist as an install permission. † Deny if no other case matches.)

Grant Permission Cases	Protection Level	SDK Level	Grant As
Grant install	Normal	Any	Install
Grant install	Signature	Any	Install
Grant legacy install	Dangerous	<23	Install
Grant upgrade*	Dangerous	>=23	Runtime
Grant runtime	Dangerous	>=23	Runtime
Deny †	-	-	-

update now all require a scan over all the other existing applications to properly adjust their permissions whenever there is a change in the set of permissions (e.g., removal of a permission). Additionally, Android's permission granting scheme changed drastically with the introduction of runtime permissions. We aim to reflect all of these changes in our formal model. It is important to note that the order of statements in the presented predicates do not affect their correct operation since Alloy is a declarative (rather than imperative) language.

Grant Permissions. In contrast with the install permission model where permissions can be granted only at installation, in the runtime permission model, depending on the protection level and the app's target SDK level, permissions can be granted as either install or runtime permissions. Permissions with protection level normal and signature are always granted as install permissions, whereas for dangerous permissions, the behavior changes based on the target SDK level of the app being installed.

Table 5.3 shows the cases that can happen for when granting permissions. Each case will add/remove "stateful" permission objects for this app. As explained in Section 5.3, we observed implementation flaws in this part of the Android source code which make the aforementioned attacks possible and we mirrored the same erroneous behavior in our Alloy predicate for granting permissions (`grantPermissions`). For example, when denying "dangling" permissions to apps, we skip to revoke runtime permissions and only revoke install permissions as it is currently implemented in Android. Also, when a custom permission is updated from normal to dangerous protection level, we treat this as an app SDK update—just as Android mistakenly does—and automatically grant the dangerous permission without user's consent. Note that our final formal model corrects these and other problematic issues according to Cusper's design. Listing 5.4 illustrates this operation; an example to how an individual case are handled can be found in Listing A.3 in Appendix A.

Installation. As a precondition to the `install` operation, the app being installed should not

Listing 5.4: Granting permissions in the Alloy model

```
pred grantPermissions[app: Application, t, t': Time] {
  all pname: app.usesPerms.t' |
  pname in (Device.builtinPerms + Device.customPerms.t').name ⟹
  (let p = findPermissionByName[pname, t'] {
    p.protectionLevel = Normal // Case GRANT_INSTALL (normal)
      ⟹ grantInstallCase[p, app, t, t']
    else // Case GRANT_INSTALL (signature)
      p.protectionLevel = Signature and
      (verifySignatureForCustomPermission[p, app, t'] or
      verifySignatureForBuiltinPermission[p, app])
        ⟹ grantInstallCase[p, app, t, t']
    // ...other cases (grant runtime etc.)
    else // Case GRANT_DENY (deny permission)
      no pd: PermissionData | pd.perm.name = pname and pd in
         app.permissionsState.t'
  })
  else //permission doesnt exist, evoke (wrongfully only install perms)
    let pd = getPermissionData[pname, app, t]{
      hasPermissionData[pname, app, t]
      and pd.isRuntime = True
      and pd.perm.protectionLevel = Dangerous
        ⟹ pd in app.permissionsState.t'
    else
      no pd: PermissionData | pd.perm.name = pname and pd in
         app.permissionsState.t'
    }
  // make sure app cannot be granted unrequested permission
  no pd: app.permissionsState.t' |
    pd.perm.name not in app.usesPerms.t'
}
```

exist on the device and the list of apps after the operation is completed should consist strictly of all the apps before installation augmented by the new app. As a result of installation, custom permissions of this app will be added to the device. Just as it is currently handled in Android, our Alloy predicate for installation does not allow an app to declare a custom permission which has the same name as an existing permission on the device. Custom permissions that are declared to be part of system permission groups also get added to the respective groups. Finally, the permissions requested by the app will be granted to the app without affecting the permissions granted to other apps, that is "stateful" permission objects should not change.

Uninstallation. The `uninstall` operation removes an existing app and its custom permis-

Listing 5.5: Uninstall operation in the Alloy model

```
pred uninstall[t, t': Time, app: Application] {
   app in Device.apps.t // precondition
   // remove app from list
   Device.apps.t' = Device.apps.t - app
    // remove custom perms defined by app
   Device.customPerms.t' = Device.customPerms.t - app.declaredPerms.t
   all a : Application - app | grantPermissions[a, t, t']
   // remove permissions from permission groups
   all pg: Device.builtinPermGroups, p: Permission |
      p in pg.perms.t and p not in app.declaredPerms.t'
         ⟹ p in pg.perms.t' else p not in pg.perms.t'
}
```

sions from the device and it readjusts the permissions granted to other apps in case there is a change in the set of custom permissions. In order to achieve this, `grantPermissions` is executed for all the apps on the device to reassign permissions and make sure that apps will be revoked the custom permissions of the removed app. This is a new behavior introduced by Google as a response to the previous bug reports regarding the issues with custom permissions. Listing 5.5 demonstrates uninstallation in Alloy.

Update. We introduce the update operation in our model since this operation is necessary to demonstrate the *Custom Permission Upgrade* vulnerability. Similar to `uninstall`, if a custom permission defined by this app is being removed from the manifest file with the update, we invoke `grantPermissions` for all other apps on the device in order to revoke this permission. Additionally, `grantPermissions` is executed for the app being updated to readjust its granted permissions, regardless of any change on the set of permissions. Permission groups are also readjusted such that the permissions removed from the app are also removed from their respective permission groups and the newly-added permissions are added to their respective groups. Listing 5.6 demonstrates the update operation in Alloy.

5.5.3 Correctness of Cusper

In order to verify the correctness of a proposed model, one needs to first compile a set of fundamental properties that need to be satisfied by the model under all conditions. Then, we need Alloy assertions, which are sanity checks to verify that this model behaves as expected with respect to these properties.

All security properties that had to be satisfied by the original Android permission model should also be satisfied by Cusper. Here, we focus on the properties that were violated by

Listing 5.6: Update operation in the Alloy model

```
pred update[t, t': Time, app: Application] {
  app in Device.apps.t // precondition
  Device.apps.t' = Device.apps.t
  // 1. Fix custom permissions on the device
  Device.customPerms.t' =
    Device.customPerms.t - app.declaredPerms.t + app.declaredPerms.t'
  // 2. Update all other apps if a perm is removed
  anyPermissionRemoved[t, t', app] ⟹updatePermissions[Application - app,
     t, t']
  else
    all a : Application - app | a.permissionsState.t' = a.permissionsState.t
  // 3. Regrant permissions for the current app
  grantPermissions[app, t, t']
  // 4. Adjust permission groups
  adjustPermissionGroups[app, t, t']
}
pred updatePermissions[apps: set Application, t,t': Time] {
  all app: apps | grantPermissions[app, t, t']
}
```

the original model. Our observation is that the new classes of vulnerabilities we discussed in Section 5.3 are made possible because of the violation of two fundamental security properties that should always hold on the Android runtime permission model: 1) *dangerous runtime permissions should never be granted without user interaction*, 2) *there should never be an unauthorized application component access*. The first property means that a dangerous permission should only be granted with the user's approval for when the app's target API level is equal to or more than 23. The second one suggests that an app cannot access another app's components if it does not have the right permission for it. For example, if an app component is being protected by a signature custom permission, only the applications that possess the same signature as this app should be able to access the component.

In order to verify our observation, we built Alloy assertions of fundamental security properties and showed that the original model indeed does not satisfy the two aforementioned properties as the Alloy analyzer is able to produce counterexamples for both assertions indicating the violation of these properties. These correspond to the attack instances we have previously described. However, when the permission model is augmented to describe Cusper, we show that all the security properties are always satisfied, formally verifying the correctness of our design.

5.6 SYSTEM EVALUATION

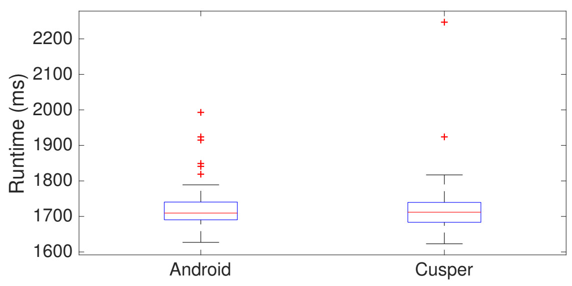
In the previous section, we verify the correctness of the formal model of Cusper which provides confidence regarding our design decisions. Next, we want to generate evidence regarding the practicality of Cusper's respective system implementation. Toward this end, we empirically evaluate our implementation of Cusper on Android, with respect to (a) its ability to thwart the specific attacks we presented and (b) its performance overhead incurred in the affected Android operations.

Effectiveness. To evaluate the effectiveness of Cusper, we carried out the two attacks we mentioned in Section 5.3 on Cusper-augmented Android and showed that both attacks fail.

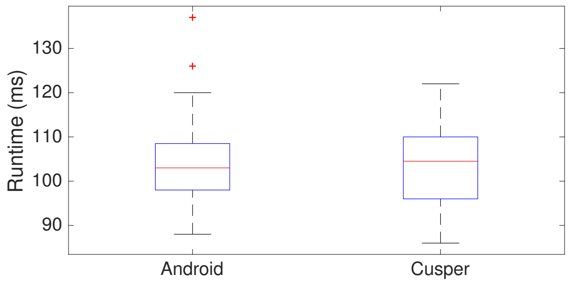
First, we attempted the *Custom Permission Upgrade* attack on Cusper-augmented Android and verified that the attack could no longer succeed. The user is correctly being consulted to grant the pemission by the system once a permission declaration changes from a normal protection level to dangerous. Moreover, we verified that a third-party app can neither assign a custom permission in a system permission group, nor declare a custom permission group using the system permission group naming convention. At the same time, normal operations of benign third-party and system apps are preserved.

With respect to the *Confused Deputy* attack, using the the apps mentioned in Section 5.2 (i.e., Skype, CareZone) as well as other real-world apps, we verified that the attack can no longer succeed while again utility is preserved with Cusper. We further tested that permission revocation happens correctly when the declarer app is uninstalled. We also verified that declared custom permissions are prefixed by a hash of the app developer's signature, and the same happens for the custom permissions used to protect app components. Finally, we tested that granting normal and signature permissions at installation time, granting dangerous permissions at runtime, and using the permissions to access protected app or system components, happen correctly; hence, we do not break any utility.

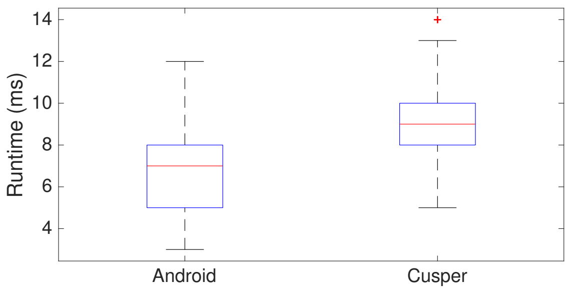
Efficiency. In evaluating the performance of our system, we focused on the operations affected by our modifications. These include the app install operation, the app uninstall operation, runtime (dangerous) permission granting, and permission enforcement. We did not include our evaluation for the app update operation as its performance is similar to that of app install. We use a Nexus 5 phone running Android 6.0 (android-6.0.1_r77) for all our experiments. According to a previous study, Android users have on average 95 apps [86] installed on their devices. In addition, according to our prevalence study in Section 5.2, apps create one custom permission on average. In order to evaluate Cusper under realistic conditions, we mimic this average case in our experiments and make sure the device contains 100 custom permissions along with all of the system permissions.



(a) App Install



(b) App Uninstall



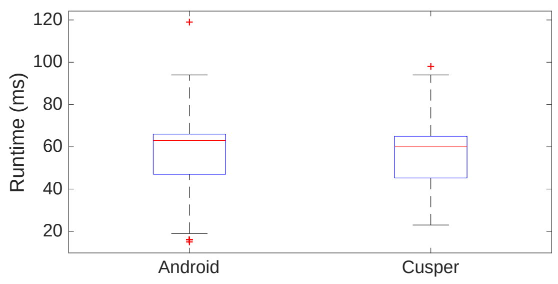
(c) Grant Dangerous Permission

Figure 5.1: Performance evaluation of Cusper for installation, uninstallation and runtime (dangerous) permission granting.

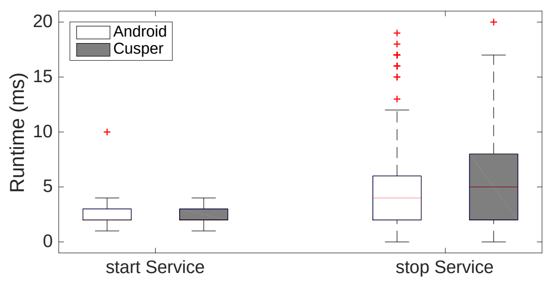
In our *app install* and *app uninstall* experiments, we used the *Android Debug Bridge* (adb) to install and uninstall an app of size 1.2 MB 100 times. The app declares a custom permission, with `protection-level` dangerous, uses the permission, and declares a service which is protected by that permission. We instrumented the `installPackageAsUser()` method in the `PackageManagerService` class to get the start time of app installation. We got the end time at the point before the system broadcasts the `ACTION_PACKAGE_ADDED` intent indicating the completion of the package installation. For app uninstallation, we instrumented the methods `deletePackage()` and `deletePackageX()` to get the start time and end time respectively. Figure 5.1a and Figure 5.1b illustrate our results.

We compared our system with the unmodified Android version (*Android*). During installation, our system performs checks during parsing, performs the permission translation, and handles the permission revocation. While parsing, it checks and stores whether a permission definition is for a custom permission and it enforces the permission group checks. Then, it parses the in-memory meta-data of an app to perform a custom permission translation. Nonetheless, as shown in our evaluation, the performance overheads are indeed negligible: there is no statistically significant deviation between Cusper and the original version.

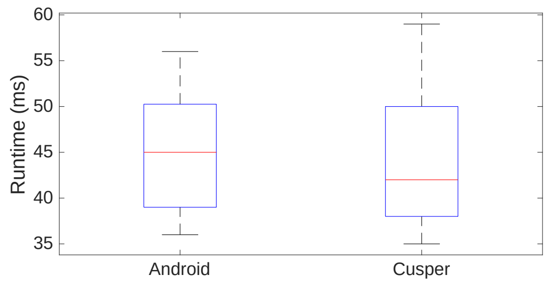
In addition, we evaluated the operation of granting a dangerous permission at runtime. We used an app which requests a custom permission previously defined in the system. Note that



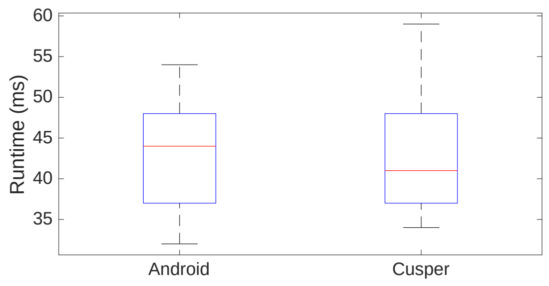
(a) Activity



(b) Service



(c) Broadcast Receiver



(d) Content Provider

Figure 5.2: Performance evaluation of Cusper for component access.

this is a process which involves user interaction: the system pops up a dialog box asking the user to grant or deny the permission request. We automated this process and ran this experiment 100 times. However, to avoid the unpredictable temporal variable of user interaction, we do not count the time between the display of the dialog box and the time the dialog box is removed. Our evaluation instrumentation is deployed in the `GrantPermissionsActivity` class. Figure 5.1c summarizes our results. Evidently, Cusper does not incur any distinctive overhead.

Finally, we evaluated the performance of permission enforcement for custom permissions. For this case, we show performance results for accessing permission-protected app components of all kinds (i.e., activity, service, broadcast receiver, and content provider) in Figure 5.2. As can be seen, Cusper indeed incurs negligible overhead for all types of component invocation operations that require permission checks.

In summary, our modifications to the Android system are shown to have no perceivable performance overhead while they greatly strengthen the security of the Android OS.

5.7 SUMMARY

In this chapter, we investigated the Android runtime permission model and identify design flaws in custom permissions that can open up ways for adversaries to escalate their privileges to obtain unauthorized access to app components and platform resources. In order to systematically fix these flaws, we proposed a defense mechanism, Cusper, that provides separation of trust between system and custom permissions and introduces an internal naming convention for custom permissions to effectively track their origins. To show the correctness of our approach, we first constructed a formal model of the Android runtime permission model using Alloy specification language and formally proved the existence of the vulnerabilities in this model. Then, we leveraged this formal model to show that Cusper satisfies the fundamental security properties that were previously violated due to the custom permission vulnerabilities. Our evaluation of Cusper on Android showed that Cusper effectively fixes the existing vulnerabilities while inducing minimal overhead.

CHAPTER 6: OVERPRIVILEGE IN RUNTIME PERMISSIONS

Android offers apps certain interfaces to communicate with the system to utilize sensitive data and platform resources that are made available by the platform and it utilizes a permission system to regulate access to these resources. Android 6.0 introduced runtime permissions to provide users with a finer-grained permission model that offers more contextual information to help them make more conscious security decisions. In this chapter, we demonstrate a new class of attacks on Android runtime permissions that constitute a significant threat to the security and privacy of Android users. We present the internals of our attacks and demonstrate their feasibility with user studies. We also demonstrate how we found existing security mechanisms to fall short in preventing such attacks.

6.1 INTRODUCTION

Android's permission model has been subject to much criticism due a range of issues including its coarse granularity [87], side-channels to permission-protected resources [88, 89, 90, 91], and design issues with custom permissions [51, 6], all of which have been addressed by Google avoiding any significant structural changes to the permission model. Another aspect of this permission model that was highly reprehended was its static scheme, under which permissions were granted to applications at installation time and they could never be revoked throughout the course of the lives of the apps. Previous work has investigated the effectiveness of Android' install-time permission model and has concluded that users would benefit from having the ability to revoke permissions on demand [16]. Finally taking a step towards bringing more flexibility into such security decisions, Android underwent a major update in its security model for permissions and switched to a dynamic permission model (i.e., runtime permissions) where permissions would be requested from the user dynamically and could be individually revoked by the user at any time. This was primarily done in order to provide a notion of context (i.e., what triggered the permission dialog) to users, so that they would have some awareness of how the permissions is going to be utilized and could make more informed decisions. It is clear that this runtime permission model is a major step towards achieving least privilege on the Android platform as it aims to grant apps the least amount of permissions to perform their functionality at any time based on user preferences.

As Android's permission system forms the backbone of access control on the platform resources and sensitive user data, any issue in this model have the potential to deeply threaten the security and privacy of all mobile device users. In particular, the runtime permission model warrants certain security guarantees for its correct operation. First, the model provides a *contextual guarantee* that ensures users will always be given the necessary contextual information to make informed decisions. This is achieved by allowing only foreground apps to request permissions in an attempt to make sure contextual cues will always be seen by the user at the time of requests. Second, the model provides an *identity guarantee* that ensures that the user is well-aware of the identity of the app owning the current context. This is realized by clearly displaying the name of the requesting app in the permission dialogs to inform the user about the identity of the app in question. In this work, we have identified that both of these security guarantees are intrinsically broken due to implicit assumptions that are taken for granted by the platform designers and we show how our findings can be used to wreak havoc on runtime permissions. First, we show that a background app is still able to request permissions by utilizing existing Android APIs to freely move within the activity stack while disguising itself via invisible overlays. Second, we observed that the naming scheme utilized in the permission dialogs assumes that the app name will uniquely identify a single installed app on the user device for its secure operation; however, the Android ecosystem does not readily enforce any rules on these names. As a consequence, apps can now spoof the names of other apps or use internal names that are completely irrelevant to their Google Play listing names but have the ability to refer to multiple apps at once with mischievous tricks. The combination of these findings indicate a major *identity crisis* for runtime permissions; background apps can now request permissions while impersonating the foreground app, a catastrophic scenario which we call *false transparency attacks*. Our attacks constitute a serious security hazard for Android's runtime permissions because 1) they allow background apps to stealthily obtain and use permissions, which is clearly undesired by Android and defeats the purpose of using runtime permissions to provide context, and 2) they create an opportunity for malicious apps to exploit user's trust in another, possibly a high-profile app to obtain permissions that they would normally not be able to acquire, breaking the security guarantees of Android's permission system all together.

Our attacks serve as a platform for adversaries to easily obtain *any* set of dangerous permissions by exploiting user's trust in other apps. As in all phishing attacks, it is important that our attacks appear realistic and do not arouse any suspicion in the user in order to profitably mislead them to grant permissions. This requires the permission dialogs triggered by the adversary to appear plausible, as users have a strong tendency towards denying permission requests that seem irrelevant to the app's use [48, 16]. We achieve this by first ensuring there is an app on the foreground at the time of a request in order not to make confusing permission requests that do not provide any context. Then, in order to *only* request permissions that are relevant to the functionality of the victim app (e.g., navigation app asking for location but not SMS etc.), we infer the identity of the foreground app and dynamically adjust our attack to only prompt the user for a permission that is already declared in this app's manifest file, as this is a strong indication that the permission is necessary for some utility provided by this app. Android previously provided APIs (e.g., `getRunningTasks()`) as well as public resources (i.e., the proc filesystem) that could be easily utilized to infer the identity of the foreground app; however, most of these have been gradually deprecated or shutdown by Google in order to address concerns regarding phishing on the Android platform. More recent work has employed machine learning techniques to automatically identify the remaining public parts of the procfs (ProcHarvester) [56] or public Android APIs (ScanDroid) [57] that can still be used to infer the foreground app by analyzing the time series information collected from these sources, with a very high success rate on event more restricted Android versions (7.0 and 8.0). In our work, we build on the ProcHarvester system and significantly modify it to infer the foreground app in real time and under realistic scenarios. Our experiments show that we are able to infer the foreground app with an 85% accuracy within an acceptable time frame.

Additionally, we hypothesize that users might be intrinsically more vulnerable to our attacks under certain circumstances due to their lack of complete understanding of runtime permissions as well as their previous interactions with apps for permissions and we can utilize this knowledge to improve the success of our attacks. To this end, we performed an IRB-approved user study with 60 participants we recruited from mTurk in order to understand the effects of these factors on our attacks. Our findings suggest that even though users generally have a good understanding of the basics of runtime permissions, they are very much confused about intricate and complex details of this model. In particular, users are not aware of the security guarantees provided by runtime permissions, making them especially vulnerable to our attacks. Furthermore, users seem to exhibits consistent behavior in terms of granting permissions to apps; if they have previously granted a permission to an app, they tend to repeat this decision for a future request for the same app-permission pair. Hence, adversaries can utilize this consistency in granting behavior to improve the success of their attacks. We have shown that a user base of 87% can be successfully mislead to fall victim to our attacks with this strategy. Additionally, none of our participants reported any suspicious behavior at the time of our attacks, simply not noticing an attack has taken place.

As we have previously mentioned, Android's main strategy for defending against phishing attacks has been revolving around blocking access to certain APIs and public resources that provide a medium for obtaining the identity of the foreground app to successfully carry out such attacks. However, we argue that dwelling more on similar approaches that aim for identifying remaining sensitive parts of procfs as well as other potential side-channels and

Table 6.1: Android Distribution Dashboard [73]

Code name	Version number	API level	Adoption
Marshmallow	6.0	23	16.9%
Nougat	7.0	24	11.4%
	7.1	25	7.8%
Oreo	8.0	26	12.9%
	8.1	27	15.4%
Pie	9.0	28	10.4%
Total			**74.8%**

closing them down to prevent phishing will prove to be infeasible in the long run due to utility reasons. Hence, we advocate for a more elegant and systematic approach to phishing on Android by targeting the main enabler of these attacks: the ability to start activities in the background. In fact, we observe that Google finally decided to move towards this drastic but clean approach in the most recent Android release, Android Q (released as a beta preview version as of the time of writing). When implemented correctly, this approach should indeed be effective against all phishing attacks. However, we identified ways to evade this security mechanism to still be able to silently start activities in the background; hence, we are still able to launch our attacks on this latest Android release with some modifications.

6.2 RUNTIME PERMISSIONS IN THE WILD

Our attacks constitute a significant threat to the security of runtime permissions. Here, we investigate the adoption of runtime permissions to demonstrate the extent of our attacks.

First, we investigate the adoption of Android versions that support runtime permissions (6.0 - 9.0). As reported by Google, the cumulative adoption of these Android versions is 74.8%, as shown in Table 6.1. This means that the majority of users are using Android devices that support runtime permissions and are vulnerable to our attacks by default.

Next we investigate the apps that adopted runtime permissions in the wild. For this purpose, we collected the 80 top free apps of each app category on Google Play store (with some failures) and obtained a final dataset with 2483 applications. We collected this dataset in December 2018, when runtime permissions had already been released for a few years. Table 6.2 summarizes our results. 83% of the apps in our dataset have a target API level 23 or more, indicating that they utilize runtime permissions. Out of these apps, 85% of them (71% of all apps) request at least one permission of protection level dangerous. This shows that runtime permissions are highly adopted by app developers and users are already

Table 6.2: Adoption of permission models and use of dangerous permissions by apps.

	Runtime permissions	Install-time permissions
Requesting dangerous permission	1755 (71%)	357 (14%)
Not requesting dangerous permission	309 (13%)	62 (2%)
Total	**2064 (83%)**	**419(17%)**

very accustomed to dealing with runtime permissions as the majority of the apps do ask for permissions at runtime. This might indicate a habituation effect, positively impacting the success of our attacks.

6.3 ATTACKING RUNTIME PERMISSIONS

According to Android's new runtime permission model, dangerous permissions are requested dynamically by prompting the user with a permission dialog. In this model, only the foreground apps are expected to request permissions as a means to ensure users will always be provided with the essential contextual cues (i.e., the triggering UI) at the time of the request. Additionally, the system ensures that the user will be made aware of the app owning the current context (and requesting the permission) by additional indicators, such as the app's name in the permission dialog. These security guarantees are provided by the runtime permission model to reliably and securely deliver contextual information. In our work, we discovered that there are certain ways to break these security guarantees by invalidating the implicit assumptions they rely on for their correct operation. By utilizing our observations, we build phishing-based privilege escalation attacks in the context of runtime permissions. Our attacks, that we call the *false transparency attacks*, allow an adversary to obtain runtime permissions from the background while impersonating other, possibly more trustworthy apps. This is clearly a violation of the security guarantees promised by Android permissions, as a *false* principle obtains a privilege under an *illegitimate* circumstance. In this section, we will discuss the essential security guarantees that are promised by the runtime permission model, how they can be broken to launch phishing-based privilege escalation attacks on runtime permissions, and the internals of our attacks in detail.

6.3.1 Threat Model.

We assume an adversary that can build Android apps and distribute them on app markets, such as Google Play (GP) store; however, the adversary is not necessarily a reputable developer on GP. Their goal is to obtain a desired set of dangerous permissions, which is relatively difficult to achieve for non-reputable app developers as shown in previous work [48]. To this end, the adversary provides an app with some simple and seemingly useful functionality (e.g., QR code scanner etc.) to lure the users into installing the app. Once the adversary's app is installed on ap victim's device, it will impersonate high-profile apps in order to stealthily obtain dangerous permissions from the background.

6.3.2 (Breaking) The Essential Security Guarantees of Runtime Permissions

Runtime permissions strive to provide contextual information to users to help them make conscious decisions while granting or denying permissions. In order to reliably and securely deliver this contextual information, Android warrants some security guarantees: 1) users will always be provided with the necessary contextual information, 2) users will be informed about the identity of the app owning the current context at the time of a permission request. Here, we discuss how these guarantees rely on the validity of certain implicit assumptions and present the ways we discovered that can be used to invalidate these assumptions to consequently break the security guarantees of runtime permissions.

Contextual Guarantee: Users should always be provided with context during requests by allowing *only* foreground apps to request permissions. The runtime permission model strives to provide users increased situational context in order to help them with their permission decisions. Currently, Android provides contextual information to dynamic permission requests in the form of graphical user interface; that is, when the user is presented with a permission dialog, they have the knowledge of what was on the screen and what they were doing prior to the request to help them understand how this permission might be used by the app. In order to ensure users are *always* provided with contextual information at the time of permission requests, Android allows permissions to be requested *only* from the context of UI-based app components such as activities and fragments and the requesting component has to run on the foreground for this purpose. The implication of this is that only foreground apps will be able to request permissions. In our work, we discovered that this assumption is conceptually broken as we can utilize existing features offered to developers by the Android platform to enable background apps to also request permissions. To elaborate, Android provides mechanisms that give apps the ability to move up and down within the activity stack. In addition, apps can be made *transparent*, simply by setting a translucent theme. By combining both of these mechanisms, a transparent background app can be surreptitiously brought to the foreground for a limited amount of time, only to immediately request a permission and then it can be moved to the back of the activity stack once the request is completed by the user. This way, a background app gains the ability to request permissions without providing any *real* context to the user as the user is presented only with the context of the legitimate foreground app due to the transparency of the background app that is overlaid on top of it. It is important to note here that permission requests do freeze the UI thread of the requesting app so nothing will happen if the user clicks on the screen to interact with the app itself. This way, the user will not have the opportunity to detect the current foreground app is not what app they think it is (i.e., it is actually the background app in disguise) by simply trying to interact with the app.

Bianchi et al. discusses some of the ways they discovered how a background app can be moved to the foreground [92]. Here, we discuss some of these techniques that were previously discussed as well as some other ways we discovered that could achieve the same goal.

• *startActivity API.* Android provides the `startActivity` to start new activities, as the name suggests. According to Bianchi et al., using this API to start an activity from a service, broadcast receiver or a content provider will place the activity at the top of the stack if `NEW_TASK` flag is set. However, we found that simply calling `startActivity` without setting this flag in these components also seems to achieve the same thing in recent Android versions. In addition, they found that starting an activity from another activity while setting the `singleInstance` flag also places the new activity on top of the stack. We found that setting this flag is not necessary to achieve this anymore, even simply starting an activity from a background activity seems to bring the app to the foreground.

• *moveTaskTo APIs.* `moveTaskToFront()` API can be used to bring an app to the foreground. This API requires the `REORDER_TASKS` permission, which is of protection level normal and is granted at installation time without user approval. In addition, `moveTaskToBack()` API can be used to bring apps to the back of the activity stack. In this case, we observed that the app continues running in the background (i.e., `Activity.onStop()` is not called unless the activity actually finishes its job).

• *requestPermission API.* According to Android's official developer documentation, `requestPermission(String[], int)` API can only be called from components with user interface such as activities and fragments. This is in line with the main goal of runtime permissions, to provide users a sense of situational context before they make decisions regarding permissions. A similar version of this API with different parameters is also implemented in the Android support API to provide forward-compatibility to legacy apps. This version, `requestPermission(Activity, String[], int)`, takes an extra activity parameter and uses the context of this activity to request the permission. This support API makes it possible to request permissions from non-UI components, such as services and broadcast receivers. In addition, if `requestPermission()` API is called from a non-UI component or from an active background activity (i.e., brought to the background by `moveTaskToBack()` API), the app will be automatically brought to the foreground for the permission request.

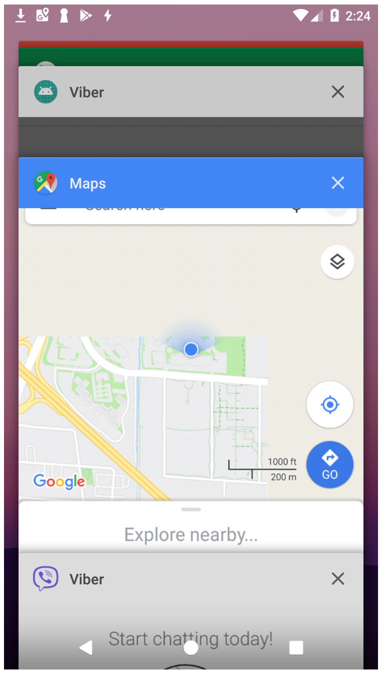
Identity guarantee: Users should be made aware of the identity of the requester via uniquely-identifiable app names in permission dialogs. Android allows apps to be started automatically via its intent mechanism for IPC without requiring user's approval or intervention. This can create an issue for permission requests since the user might not be able to tell the identity of an automatically-launched app if it were to immediately request a permission, as they have not personally started or been interacting with this app. In order to overcome this issue, Android displays the name of the requesting app on permission dialogs in an attempt to help users quickly identify the app owning the current context.

Even though this mechanism initially seems like an effective solution to app identification problem for runtime permissions, it is insufficient due to the lack of app name verification on the Android ecosystem. Each Android app listed on the Google Play store has a *Google Play (GP) listing name* that is displayed to the user while browsing this store, as well as an *internal app name* that is defined in the resource files of the app's apk and displayed when the user is interacting with the app on their device, including in the permission dialogs. Google Play store does enforce certain restrictions on GP listing names. For example, it produces warnings to developers when their GP listing name is too similar to that of another app and does not allow the developers to publish their apps in this case, in an attempt to prevent typo-squatting and spoofing that can be used towards phishing attacks. However, the same kind of scrutiny does not seem to be shown when it comes to internal app names as the Android ecosystem does not enforce any rules on these names. Our observation is that 1) the internal name of an app can be vastly different than the app's GP listing name and 2) multiple apps can share the same app name, even when installed on the same device. For example, we have successfully published an app on Google Play store, where the internal name of our app was "this app" even though the GP listing name was completely different, a case we will make use of in our attacks as we will explain in more detail later. We were also able to spoof the name of a popular app (i.e., Viber) and successfully release our app with this internal app name on GP. In short, Android does not perform any verification on the app names shown in runtime permission dialogs to ensure the identity of the requesting app matches the user's expectations.

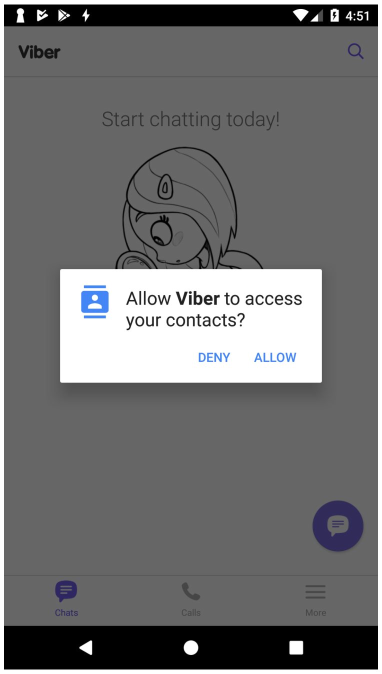
6.3.3 Wreaking Havoc on Runtime Permissions: False Transparency Attacks

By combining the inherent ability of apps to move within the task stack in disguise and Android's lack of app name verification, we built the *false transparency attacks*, where a *transparent* background app temporarily moves to the foreground while impersonating another, possibly more trustworthy app that was already in use by the user (i.e., on the foreground) and requests a permission it sees fit. After the user makes a decision to either grant or deny the permission, the attack app immediately moves to the background again in order to evade detection so that the user can continue interacting with the legitimate foreground app without noticing they have been attacked.

A demonstration of our attack including the state of the task stack before and during the attack can be observed in action in Figure 6.1. Figure 6.1a displays the task stack immediately before the attack takes place. As can be seen, Viber, a popular communication app with millions of downloads, is at the front of the task stack (shown in the bottom) and at the back of the task stack there is another app also called Viber, representing the attack app running in the background targeting Viber for permissions. Here, it is worth noting that we are showing the real content of the task stack for demonstration purposes and the attack app can in fact be easily hidden from the task stack in order to evade detection by the user, by utilizing the `finishAndRemoveTask()` API or the `android:excludeFromRecents` tag in the Android manifest file as discussed in Section 2.1.4. At the time of the attack, user will experience a user interface (UI) that is similar to the one in Figure 6.1b. Here, the app prompting the user for a permission appears to be Viber, as both the UI displayed underneath the permission dialog and the app name in the dialog strongly indicate the app to be Viber. However, the request is, as a matter of fact, triggered from the transparent attack app that surreptitiously made its way to the foreground and covered Viber. This can be observed by displaying the state of the task stack at the time of the attack, as shown in Figure 6.1c. As can be seen, the forged Viber app that belongs to the attacker is in fact at the forefront of the task stack (seen at the bottom) and the real Viber app is immediately behind it at the time of the attack, creating a confusion about the origin of the permission request for the users due to the identicalness of the shown user interface to that of Viber. Additionally, the attacker was able to spoof the internal app name of Viber in the permission dialog to further mislead the user into thinking the permission request indeed originated from Viber, as shown in Figure 6.1b. All in all, the contextual cues given to the user in this attack scenario (i.e., UI, app name, the identity of the foreground app) appear to be indistinguishable from a benign scenario where Viber is the legitimate permission-requesting app, from the perspective of device users.

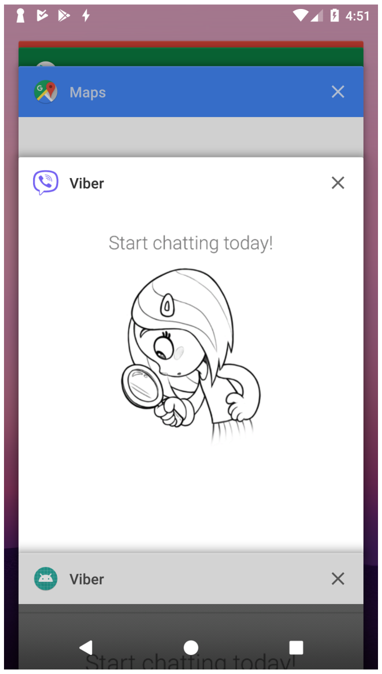


(a) Pre-attack task stack



Allow **Viber** to access your contacts?

(b) Stealthy request



(c) Task stack during the attack

Figure 6.1: Background app requesting a permission pretending to be the foreground app (Viber).

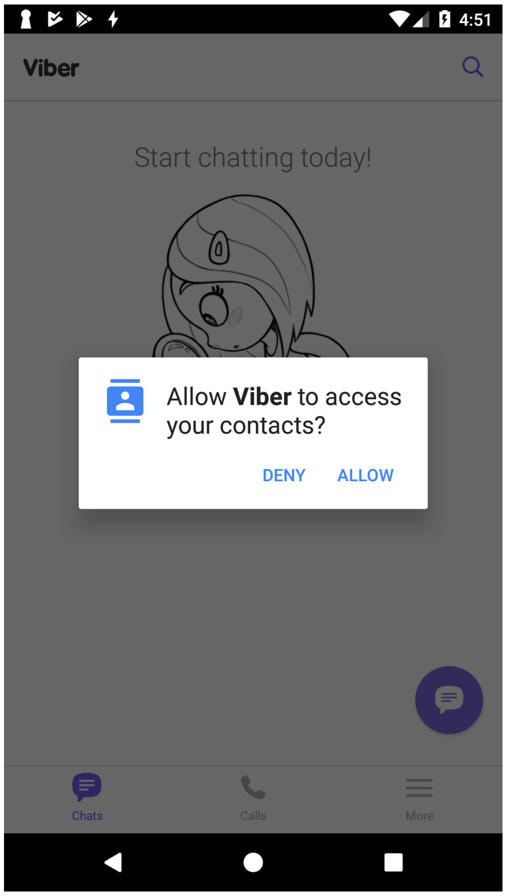
Plausible and Realistic Attacks. We intend our attacks to serve as a platform for adversaries to easily obtain *any* set of dangerous permissions by exploiting user's trust in other apps while not arousing any suspicion. With each request, the adversary is essentially exposing themselves to the user and is risking the possibility of alerting the user to be suspicious and take action accordingly (e.g., scan their apps to uninstall questionable ones). Therefore, it would be in the adversary's best interest to request permissions sparingly, only when the possibility of user granting the permission is considerably high. In order to achieve this, we utilize several techniques as we will now explain.

First, users are accustomed to be asked for permissions by an app running on the foreground. Hence, we do not request permissions when there is no app on the foreground in order not to surprise user with the permission request. For this purpose, we utilize the `getRunningTasks()` API, which previously provided the identity (i.e., package name) of the app on the foreground, but was deprecated in Android 5.0 due to privacy reasons. However, we discovered that this API still provides limited amount of information which can be utilized to detect the existence of a foreground app. More specifically, this API now outputs `com.google.android.apps.nexuslauncher` if there is no app on the foreground, indicating the nexus launcher. Else, it outputs the package name of the app calling the API (whether this app is on the foreground or not), indicating the existence of a running foreground app.

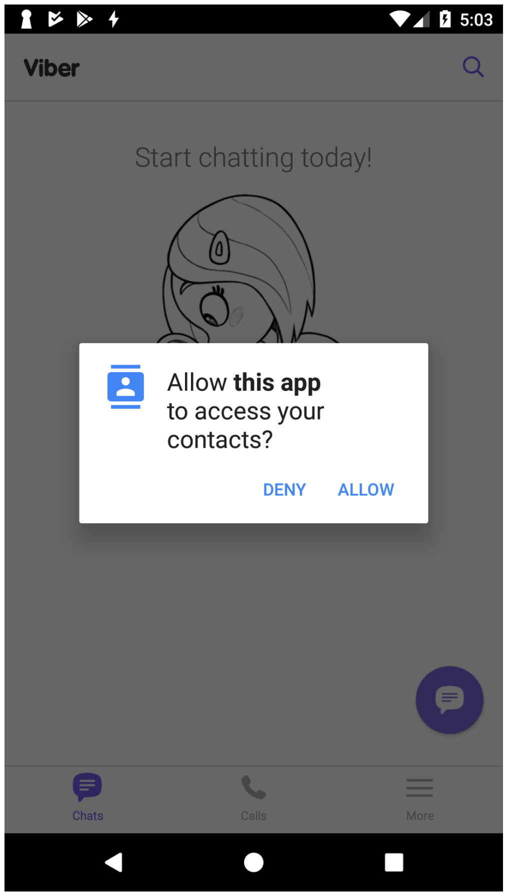
Second, previous work has shown that when users make decisions to grant or deny permissions, they consider the relevance of a requested permission to the functionality of the app. If they think the app should not require a certain permission for its operation or it should still work without that permission, they generally choose to not grant the permission [48, 16]. Taking this observation into account, in our attacks we avoid requesting permissions that are certainly irrelevant to the functionality of the foreground app because such requests will likely result in the user denying the permission. Here, we consider a permission to be relevant to the functionality of an app only if the app declares this permission in its manifest file and intends to use it. In order to identify these relevant permissions, we first need a mechanism to detect the identity of the victim app on the foreground (i.e., its package name) so that we can determine its declared permissions. For this purpose, we utilize ProcHarvester, which uses machine learning techniques to identify the public parts of Android's proc file system (procfs) that can be used to infer the foreground app on Android, even when access to procfs is mostly restricted by Google due to privacy reasons. We modify ProcHarvester to fit realistic attack scenarios and also implement real-time inference of time series on top of ProcHarvester to detect the identity of the foreground app in real time. After obtaining the package name of the foreground app, we can use this information to query `PackageManager` to obtain the permissions required by this app and request only those permissions in our attacks. The details of our ProcHarvester-based implementation for foreground app inference will be described in Section 6.4.

Additionally, we speculate that there might be other factors that can help with improving the success and plausibility of our attacks. As such, we hypothesize that users' general understanding of how runtime permissions work and their previous decisions regarding trust in victim apps for permissions might also influence their current decisions and consequently have a substantial effect on the success of our attacks. Therefore, the attacker should not only detect the foreground app to identify its relevant permissions, but also be strategic about which of those permissions should be requested by taking into account users' past interactions with victim apps in this context. In order to test the validity of this hypothesis, we performed an online user study with 60 participants from mTurk. Our study shows that even though users generally have a good understanding of runtime permissions, they are confused about the intricate details of this model, rendering them vulnerable to our attacks. In addition, users are very inclined to grant a permission again if they had previously granted it to the same app in question; that is, they tend to carry their past decision for granting a certain permission to an app into the future requests for the same app, permission pair. 62% of all participants will immediately grant such a permission on the second request without considering any other factor and 25% of all participants will grant such a permission if they think it is required for the functionality of the app, resulting in a total of 87% of all participants willing to grant a previously-granted permission. Given these results, it is reasonable for the attacker to mainly target permissions that were previously granted to the victim in order to maximize the success rate of the attacks and we build our attacks accordingly based on this observation. We will cover the results of our user study at greater depth in Section 6.5.

Apart from the functionality reasons to grant permissions, the reputation of the app developer has also been shown to be another major decision factor for users when granting permissions and this behavior seems consistent regardless of the permission being requested [48]. For this reason, it is important that the attacker utilizes victim apps that are highly-popular and have gained users' trust in order for the attacks to be successful. Therefore, we limit our victim apps to be from the top 100 free apps of Google Play (GP) store. This subset, to our surprise, contains many gaming apps with unknown developers; hence, we further filter out these apps to finally obtain 62 most popular and high-profile Android apps developed by well-known/trusted companies and developers. It is worth noting that we have also devised a way for the adversary to not have to target only a single app from this dataset and instead target *any* of these victim apps *simultaneously* by utilizing the lack of app name verification against GP listing names. Such a general attack scheme can be desirable over a scheme that targets only a single app when the attacker needs multiple permissions that can be provided by only a combination of victim apps, in line with our idea of our attacks providing an attack platform for adversaries to obtain *any* of their desired permissions. In addition, this general scheme can give the adversary more chances to deploy their attacks, as now there are multiple apps that can be targeted. To elaborate, in this general attack scheme, the attacker chooses a name that can *logically* represent any foreground app when displayed in a permissions dialog, mischievously taking advantage of the plain English interpretation of the question displayed in the permission dialog. More specifically, the adversary selects **this app** as their attack app's internal name to be displayed in the permission dialog and the question in the dialog will now read as "Allow **this app** to access your contacts?", as shown in Figure 6.2b. Clearly, this question's plain English meaning does not distinguish between apps and is capable of referring to *any* app that is *currently* on the foreground for a given permission request. We verified that such an app is accepted to the GP store and can be installed on a user device via GP without a problem. We have also verified with our user study that users seem to disregard this key piece of information in their choices as they are not at all aware of the security guarantee (i.e., identity guarantee) provided by displaying app names in the permission dialogs and they are naturally more concerned about the plain English meaning of the statements in the dialogs.



(a) Targeted attack scheme



(b) General attack scheme

Figure 6.2: Two app name options to be displayed in permission dialogs for the attacker to impersonate the foreground app (Viber).

Attack Steps. As we have explained the overall idea of our attacks and our methodologies, we will now give a detailed step by step guide to creating an identity crisis for permissions on Android.

1)*Lurk in the background.* The attack app creates a service to continuously run in the background and periodically collect information about the running apps to determine when to attack. Prior to Android 8.0, running background services could be achieved by using `Service` or `IntentService` classes. However, Android 8.0 brings restrictions to background execution of apps in order to preserve device resources to improve user experience. Background services are now killed a few minutes after the app moves to the background; hence, use of `JobScheduler` is more appropriate for our attacks [93] as JobSchedulers can indefinitely run in the background to periodically execute jobs.

2)*Chose your victim carefully.* The attack app runs our ProcHarvester-based implementation for foreground app inference while in the background to detect a victim app on the foreground. This comprises of continuously monitoring the proc filesystem and running our real-time foreground app inference algorithm to detect the app if we notice a change in the foreground. We will describe our implementation in more detail in Section 6.4.

3) *Chose your permission carefully.* Once we obtain the foreground app, we will query the `PackageManager` to obtain the requested permissions of this app and prompt the user for a permission that was granted to the victim but not to the attacker. If there is multiple such permissions, we will randomly pick one to request. Note that more intricate selection algorithms can be used to more properly pick the permission to be requested. For example, previous work has shown that microphone permission is the most denied permission and hence can be considered the most valuable from the perspective of our attack [48]. In this case, the attacker might want to prioritize the microphone permission if the foreground app can make a very good case of needing microphone (e.g., music app or communication app). However, we do not perform this kind of advanced permission selection as our main purpose is to demonstrate our attacks realistically without overly complicating our implementation.

4) *Cloak and dagger.* Once the attacker determines that a certain permission should be requested from the victim on the foreground, they will start an invisible activity from the background service, which will then be automatically moved to the foreground as we have previously explained in Section 6.3.2. After this, the attacker will prompt the user for the chosen permission from the context of this invisible activity.

5) *Leave no trace behind.* Once the user completes the permission request, the attacker will call the `moveTaskToBack()` API in order to move to the back of the activity stack to evade detection and continue running silently. This way, the victim app will be restored back to the foreground and the user can continue interacting with the victim.

6.4 FOREGROUND APP INFERENCE

As we have described in Section 6.3, the adversarial app running in the background will continuously try to infer the foreground app to determine if a known victim app is on the foreground and in that case identify any suitable permissions to be requested while pretending to be the victim app. Here, we will explain the previous efforts for foreground app inference, why they fail to work in realistic scenarios, and our implementation for inferring the foreground app in real time.

Past efforts for foreground app inference. Previously, Android offered convenient APIs, such as `getRunningTasks()`, that could be used to easily infer the identity of the foreground tasks; however, these APIs have been deprecated in Android 5.0 (API level 21) by Google in an effort to maintain the privacy of the running apps and prevent phishing attacks on the platform. This has consequently led to a search to identify other avenues that can accomplish the same task. Having inherited many features and security mechanisms from Linux, Android, too, has a proc filesystem (procfs) that provides information about the running processes and general system statistics in an hierarchical structure that resides in memory. Security researchers have discovered that Android's proc filesystem provides numerous opportunities for attackers to infer the foreground app [54, 55]. In response, Android has been gradually closing access to all the sensitive parts of the procfs pointed out by researchers in order to prevent phishing attacks. In the most recent Android versions, all of per-process information on the proc filesystem has been made private (i.e., accessible only by the process itself) and only some of the global system information have been left to be still publicly available due to utility reasons, rending the efforts to identify the foreground app virtually impossible.

More recently, though, Spreitzer et al. discovered that despite all the strict restrictions on the procfs, there are still public parts of this filesystem that initially seem innocuous but in fact can be utilized to effectively identify the foreground app by employing a relatively more complex analysis in comparison to the previous efforts. To this end, they introduced a tool named ProcHarvester that uses machine learning techniques to automatically identify the procfs information leaks (i.e., foreground app, keyboard gestures, visited websites) on potentially all Android versions, including the newer versions with limited procfs availability, by performing a time series analysis based on dynamic time warping (DTW) [56]. Then, they showed that these identified parts can be utilized for foreground app inference via a similar technique, yielding a high accuracy. In particular, ProcHarvester comprises of two main components: 1) a monitoring app that logs the public parts of procfs on the user device and 2) a server as an analysis unit (connected by wire to the phone) that collects this information from the monitoring app to first build profiles of app starts for the apps in their dataset and then perform DTW to identify information leaks. ProcHarvester currently works as an offline tool in a highly-controlled environment and is not capable of inferring the foreground app in real time, which is an absolute necessity for our attack scenario.

Real-time foreground app inference under realistic scenarios. In our work, we build on the ProcHarvester system for inferring the foreground app in our attacks. More specifically, we modified ProcHarvester to adapt to realistic scenarios and implemented real-time inference of time series to identify the foreground app. Here, we utilize the same dataset with 20 high-profile apps used in [56], to serve as the victim apps that the adversary will primarily target for permissions. We first run the original ProcHarvester implementation to create profiles of only the procfs resources that yielded high accuracy for app inference [56], for each app in our dataset. Additionally, in original ProcHarvester system, the analysis unit (server) is directly connected to the user device by wire and is collecting data from the device through this connection. However, in our case, adversaries cannot assume a wired connection to a user device as this does not constitute a realistic attack scenario. Hence, we modified the monitoring app to send continuous data to a remote server, which is running our foreground app inference algorithm in real time. This is a plausible assumption as adversaries can easily obtain the install-time `INTERNET` permission, which is of normal protection level, in order to communicate over the Internet.

Most importantly, we implemented a *real-time* dynamic time warping (DTW) algorithm to detect the foreground app. Currently, ProcHarvester can only be used as an offline inference tool, as it works based on the assumption that app launch times will be known to the tool in advance and the tool can run its DTW-based analysis starting from those launch times. However, this assumption is unrealistic in real-life scenarios as an attacker cannot assume to have a priori knowledge regarding app launch times since an app launch is either at the user's discretion or is initiated by the system or other apps via IPC. In our work, we devise a technique to identify an interval of possible values for the app start time and run DTW starting from all possible values in this interval, rather than using a single starting point as in the original ProcHarvester, to obtain the foreground app *in real time*. First, in order to obtain the starting time of an app launch, we utilize the `getRunningTasks()` API to monitor foreground changes. Even though this method was previously deprecated in the hopes of preventing phishing attacks, we observed that it still provides limited information regarding the foreground of the device. In particular, whenever there is an app on the foreground, the `getRunningTasks()` API outputs the package name of the caller app (regardless of it being on the foreground or not), and if there is no app on the foreground, it outputs `com.google.android.apps.nexuslauncher`, which corresponds to the Android launcher menu. By continuously monitoring such foreground changes, we can know if an app launch has been *completed* if the foreground state changed from "no app" to "some app", providing us the approximate *end time* (α) for the launch operation. Now, if we know the duration of an app launch event, we can subtract this from the end time to find the approximate start time of the app launch event. To identify this duration, we run an experiment on our dataset and show that app launch takes around $379ms$ on average with a standard deviation of $32.99ms$, which gives us the final range of $[\alpha - 379 - 32.99, \alpha - 379 + 32.99]ms$ for all possible app start times. For each app in our dataset, we then calculate the DTW-based distance using each of the possible values in this interval as the starting point of the analysis (with a $50ms$ gap between each point) and take their average to obtain the final distance.

Lowest of these distances corresponds to the foreground app.

In order to evaluate our foreground app inference implementation[1], we conduct experiments where we launch each of the 20 apps in our dataset 10 times and report the overall accuracy and performance. Our experiments indicate a success rate of 85% for correctly inferring the foreground app with our algorithm. In addition, we find the total time to estimate the foreground app (after its launch) to be 7.37s on average with a standard deviation of 0.98s. We consider this to be a reasonable delay for our attacks as we expect users to stay engaged with one app before they switch to another for much longer than this duration (18.9s or more on average) as shown in [94]. Since the foreground app will presumably not change during the analysis, the adversary should not have any problem targeting the identified app in their attack after this introduced delay. In addition, please note that the original ProcHarvester tool itself already needs around five seconds of procfs data to correctly compute the foreground app.

It is worth mentioning that ProcHarvester is inherently device-dependent since an app can have distinct profiles for a given procfs resource on different mobile device and this would affect the performance of foreground app inference. Hence, in order to launch a "full-blown attack" that can work on multiple mobile devices, adversaries would have to obtain the procfs profiles of their victim apps on all those devices. Here, adversaries could conveniently adopt a strategy to collect the profiles for only the most commonly-used Android devices in order to quickly cover a satisfactory user base. Note that this extra profile data should not greatly affect the performance or accuracy of the foreground app inference, as an attacker can first identify the type of the device in real time via utilizing existing tools [95] and only use the respective profiles in their analysis, avoiding DTW-based comparisons with profiles belonging to other devices in order to improve both accuracy and performance. In our work, we utilize the profiles from only one Android device (Google Pixel) as we primarily intend our attacks to serve as a proof of concept.

6.5 USERS' COMPREHENSION OF PERMISSIONS AND BEHAVIOR

Since our attack is a phishing attack at its core, it is important for it to be realistic and persuasive for the attack's overall success. We speculate that users' comprehension of runtime permissions and their past decisions in this context will play a significant role in how users perceive our attacks and impact the success of our attacks. To this end, we performed an online survey-based user study with 60 participants to more specifically understand how users

[1]We thank Jingyu Qian for helping us implement and evaluate the aforementioned modifications to the original ProcHarvester code.

decisions to grant or deny permissions are affected by these factors. Our findings suggest that Android users generally have a good understanding of the basics of runtime permissions; however, they are very much confused about the intricate details of this permission model. In particular, users are not aware of the critical security guarantees provided by the system for runtime permissions. Furthermore, users appear to display consistent behavior in terms of decisions regarding their trust in apps with permissions as they tend to carry their previous decisions to grant or deny permissions into the future permission requests. More specifically, we observed that users are particularly more vulnerable to our attacks when they have previously trusted a victim app with a certain permission and the adversary launches their attack for the same app-permission pair. In conclusion, we have observed that a user base of over 80% can be successfully misguided with our attacks by utilizing these findings and none of our participants seem to notice any suspicious behavior or suspect any foul play during the attacks.

6.5.1 Recruitment

We first obtained an IRB approval from our institution prior to the commencement of our user study. After this, we recruited 60 participants from Amazon Mechanical Turk (mTurk) to complete our online survey. Our inclusion criteria is based on user's familiarity with Android; we only recruited people who use Android as their primary phone into our study.

6.5.2 Methodology and Results

Our user study consists of an online survey where we ask participants questions to assess their knowledge of runtime permissions and also study their reactions to certain permission dialogs that may or may not correspond to our attacks to understand trends of behavior in this context. In particular, we investigate four main areas of concern in our survey: 1) users' general understanding of runtime permissions, 2) the effect of their past behavior on their current decision-making process, 3) their in-situ reaction to our attacks, 4) their perception of the security guarantees of the runtime permission model.

At the beginning of this survey, we informed our participants that they will be asked questions about their experience with Android permissions; however, to avoid unnecessarily priming them, we do not reveal that we are assessing the feasibility of our attacks and trying to identify the most favorable circumstances for our attacks. Hence, our user study still constitutes a conservative approach where users are at their best behavior in terms of security posture since they have been made aware that we are conducting a study on app permissions, a well-known security mechanism on Android. The implication of this is that our participants might be even more vulnerable to our attacks in the real world than what our survey results reveal.

It is also worth noting that we prefer conducting an online survey over a real-world study where we deploy our attacks in the wild for several reasons. To begin with, the problem we are facing here resembles a chicken and egg problem: we need to build realistic attacks to understand user behavior; however, we cannot build realistic attacks before we have some understanding of user behavior. Hence, we argue that it is reasonable to test our hypothesis in a controlled environment to identify general trends in user behavior. Furthermore, as we mentioned in Section 6.3.3, our foreground app inference technique exhibits nondeterministic behavior and it might, although not very frequently, still incorrectly guess the foreground app at times. Hence, if we were to deploy this scheme in real life, we would obtain noisy user data, skewing our understanding of trends in user behavior. This is clearly undesirable if we want to build realistic attacks that follow user behavior as closely as possible. Additionally, our foreground app inference technique is also device-dependent, requiring adversaries to fingerprint certain devices depending on their goals in terms of target user base. For example, an adversary can simply adopt a strategy where they fingerprint some of the most commonly-used Android devices to quickly cover as many users as possible, which might be sufficient for them to make profit depending on their objectives. However, our goal with this user study is to precisely understand user behavior, requiring us to correctly interpret *all* of our user data. As such, we would have to take on the improbable task of fingerprinting *all* existing Android devices for a "deployed" user study since we cannot foresee the types of devices our participants own given the current recruitment process on mTurk. Obviously, this is not only infeasible but also utterly financially-burdening for us. Therefore, in line with our main goal of providing realistic proof of concept attacks without having to build full-fledged, fully-deployable attacks, we chose to conduct an online user study to understand user behavior. Now we will share our results for each of the aforementioned areas of concern in detail.

General Understanding of Runtime Permissions. In this part of the survey, we ask participants questions that will reflect their understanding of runtime permissions in order to assess if they are intrinsically vulnerable to our attacks due to lack of knowledge in this domain.

We first ask users to self-report their level of familiarity with Android permissions. 7% of the users identify themselves as expert, 46% as knowledgeable, 42% as average, 3% as somewhat familiar, 1% as not familiar, as shown in Figure 6.3. Around 70% of the users are aware that Android used to have an install-time permission model, but now switched to runtime permissions and at least 85% of the users have used the runtime permission model. In addition, 75% of the users are aware that runtime permission model allow them to review previous decisions regarding permissions and update them. These results so far indicate that our participants generally have a good understanding of the basics of runtime permissions.

In contrast, we observe that users' answers significantly sway from the truth when we ask more intricate questions about the inner workings of runtime permissions. When asked if they agree with the statement that only foreground apps can request permissions dynamically, 55% of the users say they agree, 22% of the users disagree, and 23% state they do not know the answer. This statement is indeed true and in accordance with the contextual security guarantee promised by the runtime permission model as we explained in Section 6.3.2; hence, it is surprising that a larger majority of the users did not say they agreed with it. However, it is worth noting that this finding does not suggest that users are aware of our attack scenarios as we will demonstrate later. It simply indicates that the runtime permission model gets too confusing for users too quickly and they are not fully aware of the contextual security guarantee provided by this model. To make matters worse, when participants are asked their opinion on whether they think an app can prompt the user again for a permission that was previously granted to it, 47% agree with the statement, 30% say disagree, 23% say they do not know the answer. This statement is intrinsically false because Android, in fact, does not allow apps to re-prompt users for granted permissions; permission dialogs are simply never shown again to the users in this case. It is alarming that such a big portion of users either think this is a plausible case or they simply do not know the correct answer as this indicates they become potential victims to our attacks. It is also worth pointing out that the users who disagree with this statement and show awareness of apps' inability to re-request granted permissions still make decisions disregarding this belief and consequently fall prey to our attacks, as we will show later in this section. This might signal that users have low confidence in their level of knowledge when it comes to such intricate and complex details of runtime permissions and/or they simply have too much trust in apps or the operating system.

Effect of past behavior and trust on current decisions. In this part of the survey, we ask our participants about how their past decisions to trust apps with permissions affects their current decisions. Our hypothesis is that we might be able to utilize the knowledge of user's past behavior in this context to accordingly adjust our attack to improve its success.

Our attack scheme allows a scenario where the adversary can prompt the user for a permission while impersonating a victim app that had previously requested the same permission,

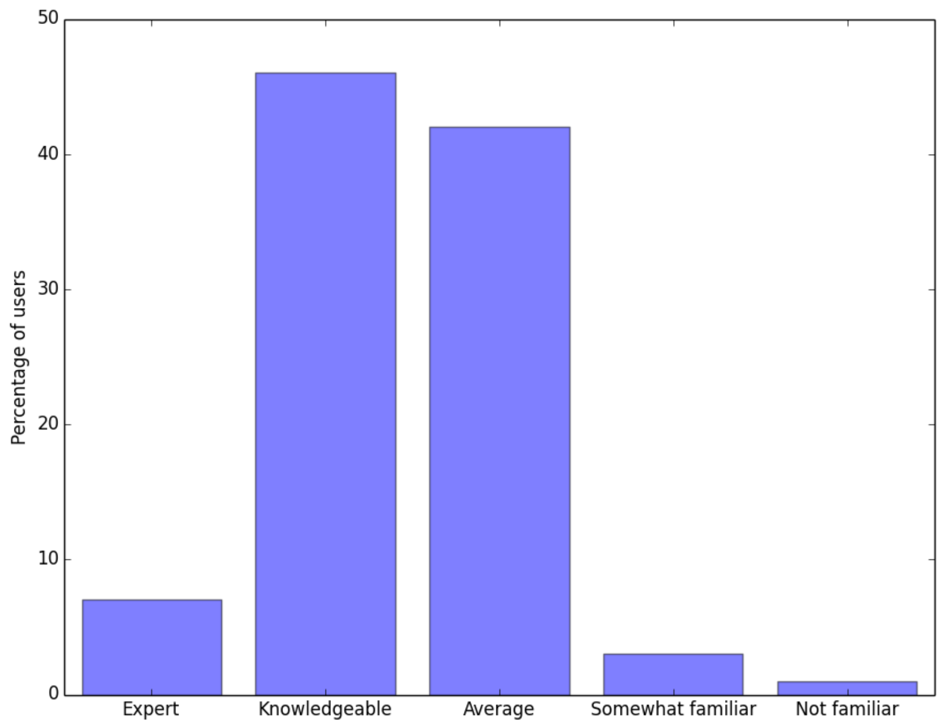


Figure 6.3: Self-reported familiarity with Android permissions

seemingly repeating the permission request for the same app-permission pair. Previously we had shown that a majority of users falsely believe such a scenario should be *always* be allowed on the Android platform regardless of what the user's previous decision was, although in fact the platform only shows permission dialogs for apps that were previously denied the permissions they are now requesting. In order to understand if we can take advantage of this wrong belief to improve the success of our attacks, we ask our participants how they would react to such secondary permissions requests. More specifically, we ask them if they would grant a permission that was previously granted or previously denied. In addition, if they state their decisions would depend on additional factors, we inquire about those with further questions. Figure 6.4 summarizes our results regarding how users react to secondary permission requests. The general trend appears to be that users are far more careful when it comes to granting previously-denied permissions. A significant portion of the users (62%) are willing to immediately grant a previously-granted permission, whereas only 27% of the users are willing to grant a previously-denied permission. In total, 28 users (47%) and 16 users (27%) stated their decision would depend on extra factors for previously-denied and for previously granted permissions, respectively. For users who made this claim, we also inquire about what these factors separately for the previously-denied and previously-granted cases. We allow our participants to list multiple factors. As can be seen in Table 6.3, users still appear to be more careful when granting a permission that they previously denied to an app. For both previously-granted and previously-denied permissions, the most important factor when granting a permission is utility; users are far more willing to grant a permission on a secondary request if they think it is necessary for the app's functionality (43% for previously-denied and 25% for previously-granted permissions). In addition, users state they will grant a previously-requested permission if they do not particularly care about it or do not consider it risky. Figure 6.5 shows the distribution of such permissions. By far the least risky permission according to our participants appears to be calendar.

Given these results, we observe that the most effective strategy for the attacker is to give precedence to requesting permissions that were previously granted by the user. This is because 62% of the users readily grant a permission that they previously granted, simply trusting their previous decision. Furthermore, an additional 25% (15 out of 60) of the users who claim to base their decisions on utility for such secondary requests might be recruited to grant a permission again for the second time, given that the attacker, as any other app, can simply utilize Android APIs to show users a permission rationale dialog with a convincing explanation of why the permission is required in order to mislead them into believing the permission is critical to the app's functionality. This means that as high as 87% of users in total could be persuaded to grant permissions in case of secondary permission requests, if the attacker were to adopt the strategy of primarily targeting permissions that were previously granted. We also cross-validate these results by presenting our participants with a storyboard of our attacks where we ask them to take action on secondary requests after making an initial decision, as we will soon discuss. As we have briefly explained in Section 6.3.3, we utilize this strategy in our attacks.

In-situ reaction to our attacks. In this part of the survey, we provide our participants with a storyboard of our attacks to study their in-situ reactions. More specifically, we ask users to role-play given descriptions of real-world scenarios along with screenshots that display permission requests which were either initiated by a popular Android app or by the attacker impersonating this victim app.

Here, our main goal is to assess users' in-situ reaction specifically to multiple requests as well as to understand if they pay attention to the app names in permission dialogs to be able to catch the attacks. For this purpose, it is more convenient for users if we described scenarios involving apps that they are familiar with and could more easily relate to. In addition, as we have discussed before, findings in previous work suggest that users take into account

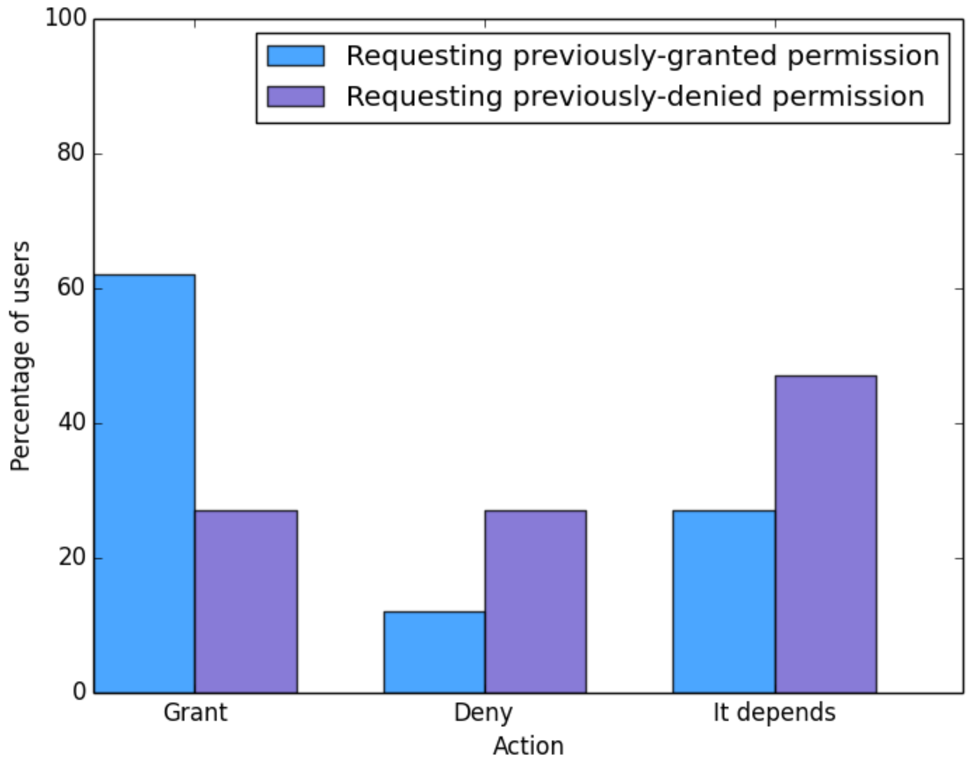


Figure 6.4: Current decision given past behavior and trust

“trustworthiness” of the apps when they are granting permissions [48]. For these reasons, as a victim app, we utilized Viber, a highly-popular messaging app with millions of downloads that allows users to communicate with their friends. Our storyboard describes a realistic scenario where the contacts permission would be requested multiple times, either by Viber or by the attacker. First, the legitimate Viber app requests the contacts permission to allow the user to message their friends. Then, after a certain time, when Viber is on the foreground again, the attacker requests the contacts permission again. For attacker’s request, instead of spoofing the real name of Viber, they use “this app” as previously shown in Figure 6.2b. We do this in order to understand if the users would fall for the generic attack that can target *multiple* victims simultaneously. Our expectation is that users will deny the permission if they notice something suspicious. In addition, we simulate the time gap between these requests by allowing the participants to answer additional survey questions in between the in-situ behavior questions. This is done in order to create a realistic situation that resonates with our attack. As can be seen in Figure 6.6, a significant portion of the users (72%) grant the contacts permission to Viber on the first request. On the second request which is a deployment of our generic attack, 67% of the users still say they will still grant this

Table 6.3: Users' preference of influencing factors when granting permissions

Influencing Factor	Number of users (previously-denied case)	Number of users (previously-granted case)
The requested permission is necessary for the app to work	26	15
The request is for a permission I do not care about or do not consider particularly risky	17	9
The requesting app is highly popular (i.e., installed by millions of users)	4	0
The requesting app is developed by a well-known company	5	2
Other	0	0

permission, 18% say they will deny the permission, and 15% of the users state their decision depends on other factors. For participants who stated they will deny the permission the second time, we ask them their reasoning to see if they declined because they might have noticed our attack. For this, we provide them a text field under the "other" category to write their comments. As shown in Table 6.4, only one participant selected "other" and they did not mention they noticed the attack in their answer. This shows that *all* of our 60 participants were oblivious to the change of app name and they did not notice our attack.

In conclusion, our attack has shown to have the potential to successfully reach a user base of 82%, by including both the users who immediately grant permissions if they previously granted them (67%) and the users who consider utility a major factor to grant permissions on the second request (15%). This result also serves as a cross-validation to the previous part which did not involve similar role-playing involving permissions dialogs but obtained similar results with more direct questions.

Table 6.4: Reasons for denying the permission on the secondary permission request

Reason	Number of users
I already granted this permission to Viber so it should not ask me again	1
I already denied this permission to Viber so it should not ask me again	5
I always decline permissions	2
Multiple requests for the same permission made me suspicious of Viber	2
Other	1

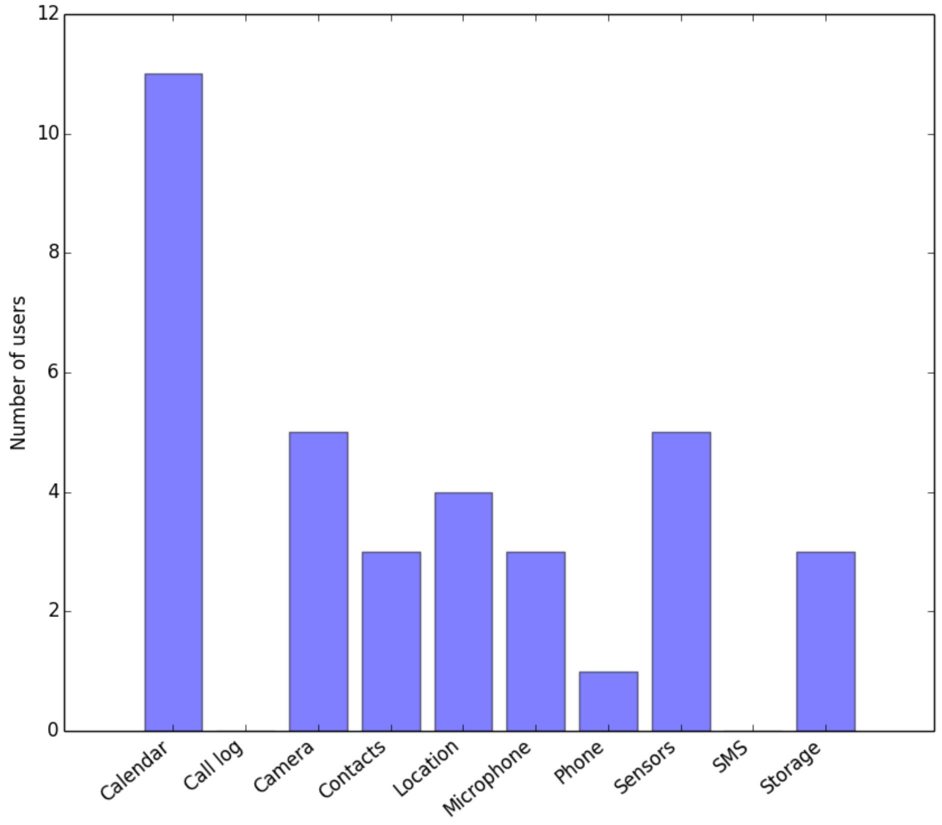


Figure 6.5: Permission groups that are considered unimportant or non-risky by users

Understanding of security guarantees. We have shown that our participants have a broken understanding of the contextual security guarantee of runtime permissions. Merely a half of our users (55%) think that permissions can be requested dynamically only by foreground apps, while 22% of the users think background apps can also request permissions, and 23% simply do not have any opinion on the matter.

We ask further questions to our participants in order to assess their awareness of the identity security guarantee provided by the app names in permission dialogs, as they all seemed to have failed to notice the unconventional app name ("this app") in the storyboard of our attacks. For this purpose, we first present them with two side-by-side screenshots of permission dialogs where the app name is "Viber" in one dialog and "this app" in the other, as shown in Figure 6.7. Then, we ask them if they think the end results of these permission requests are the same; that is, if Viber is either granted or denied the permission based on the user's response. As can be seen in Figure 6.8, 73% of users state that they think both dialogs serve to achieve the same purpose whereas 27% of users do not think this is the case. This provides further proof that most users are generally unaware of the

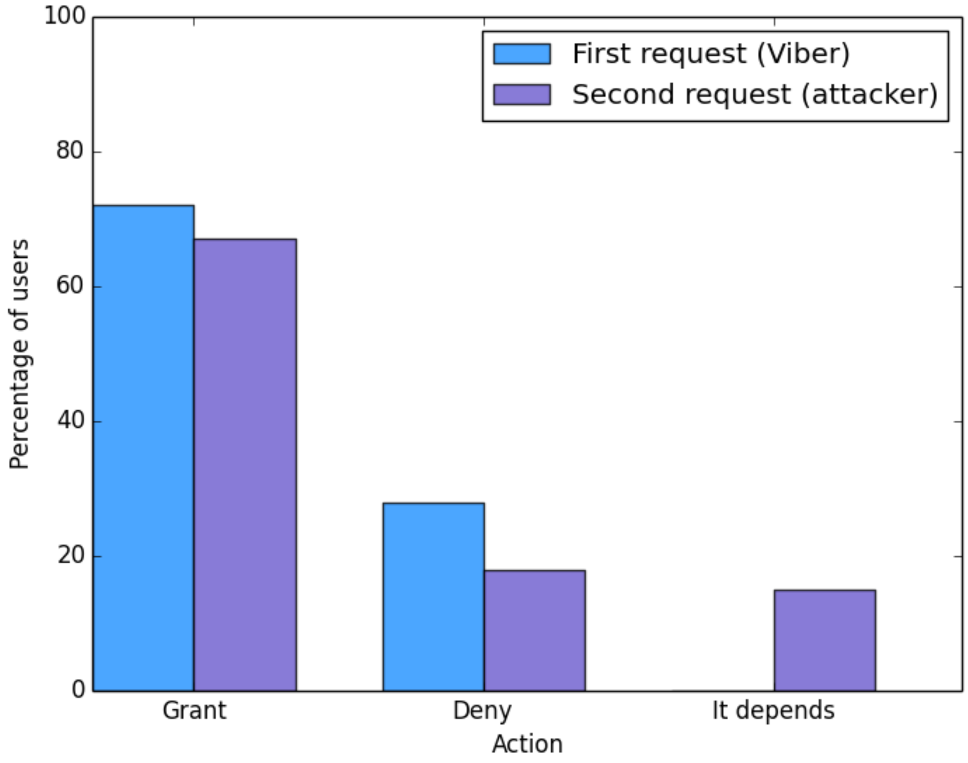


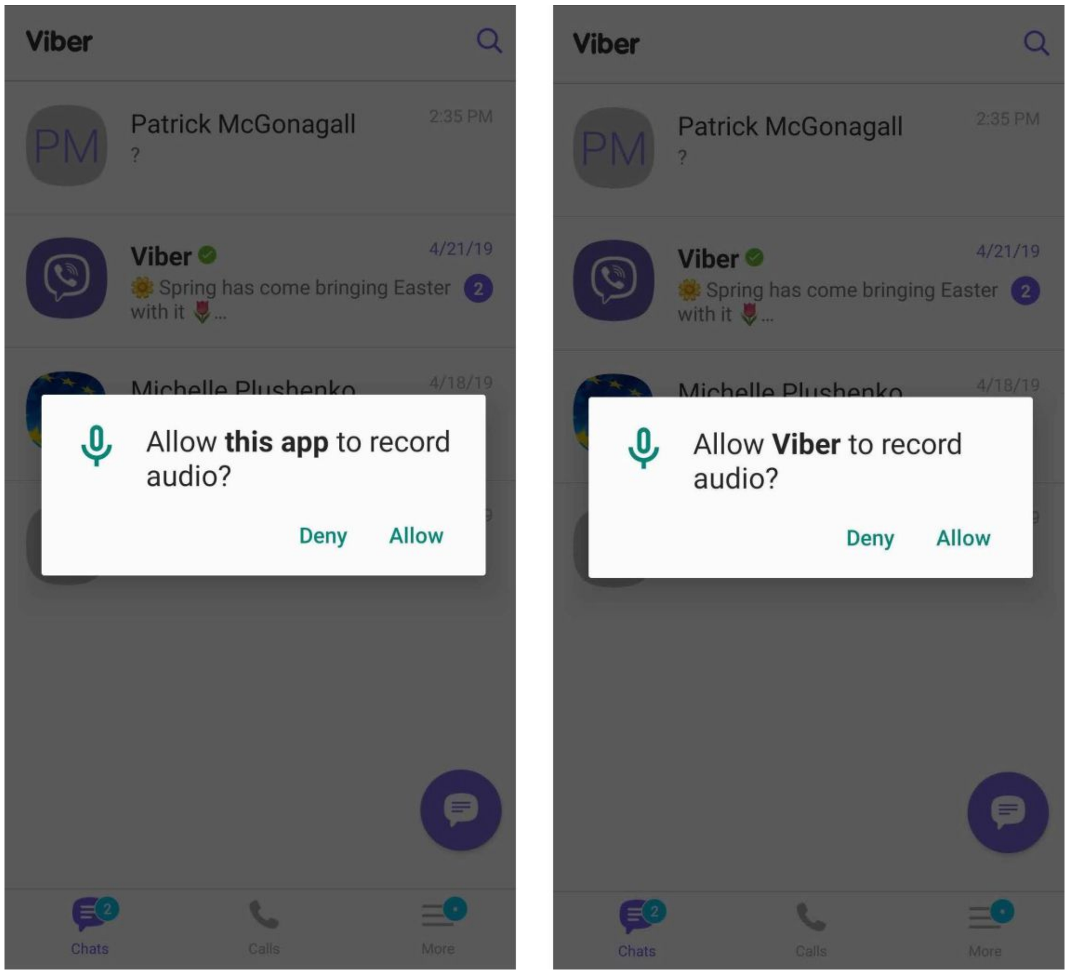
Figure 6.6: In-situ behavior with a simulation of attack

identity guarantee provided by app names in permission dialogs, as the majority still fails to recognize anything wrong in the app name even after it is specifically brought to their attention. To our attack's advantage, they seem to mostly care about the plain English interpretation of the full statement shown in the dialogs.

6.6 (BREAKING) THE IDEAL DEFENSE

Phishing attacks have been long dreaded on the Android platform as they are very hard to detect and protect against [92, 53]. In a classic phishing attack on mobile platforms, the adversary utilizes existing APIs or side channels to identify the victim on the foreground and immediately launches their own attack app that realistically spoofs components (e.g., UI, name etc.) of victim to replace the victim and lead the user to believe that they are actually interacting with the victim.

Android's way of dealing with phishing attacks has long been revolving around blocking access to certain APIs and public resources that provide a medium to obtain the necessary information (i.e., identity of the foreground app) to successfully carry out such attacks. For



Patrick McGonagall 2:35 PM
?

Viber 4/21/19
Spring has come bringing Easter with it ...

Michelle Plushenko 4/18/19

Allow **this app** to record audio?

Patrick McGonagall 2:35 PM
?

Viber 4/21/19
Spring has come bringing Easter with it ...

Michelle Plushenko 4/18/19

Allow **Viber** to record audio?

Figure 6.7: Two potential permission dialogs

example, as we have previously explained, the `getRunningTask()` API and similar APIs that provide information regarding the running apps and services on the device have been deprecated in Android 5.0. In addition, access to the proc filesystem (i.e., procfs), which provides a side channel to infer the state of the apps running on the device including the identity of the foreground app, has been gradually closed down.

However, as we have proven with our attacks, these security measures still fall short and only serve as a band-aid to a deeper problem. We argue that it is infeasible to continue putting effort into identifying and closing down all side channels that provide information about the foreground app as some of these channels are cannot be made private or deprecated due to utility reasons (e.g., monitoring apps depend on procfs to report app statistics etc.). Hence, a different approach might be necessary to address phishing on Android without compromising utility. Our observation is that the main enabler of phishing on Android is *the ability of apps to start activities in the background to replace foreground apps*. If we can stop background apps from surreptitiously replacing foreground apps, classic case of

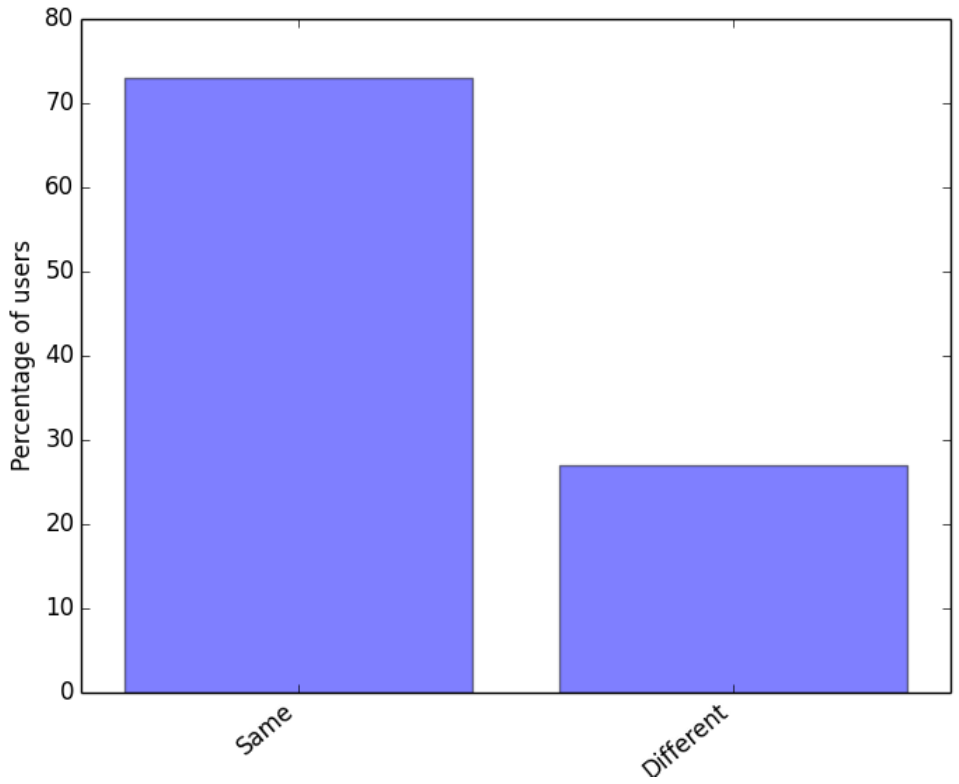


Figure 6.8: Users interpretation of two different app names in permission dialogs

phishing attacks can be easily addressed on Android.

In fact, we observed that in Android 10.0 Q (available only as a Beta pre-release as of the time of writing), Google decided to move towards taking this drastic approach to defend against phishing and eliminate it once and for all. Activity starts from background apps will now be blocked on Android 10.0 Q unless the app can justify starting an activity in the background, such as by having a service that is bound by the system or by another visible app [96]. This approach should indeed be effective in stopping the attacks if implemented correctly; however, we identified ways to evade this security mechanism to still be able to start activities in the background without satisfying any of the required conditions checked by the system to allow such an operation. Hence, our attacks will still work on Android Q.

In particular, we discovered that there are two ways that we can start activities in the background without getting blocked by the system. As we have previously explained, Android 8.0 brings restrictions to the background execution of apps and encourages the use of JobScheduler to be able to run background services efficiently. Old `Service` and `IntentService` APIs can be still used to create background services; however, such services will be short-lived as they are killed within mere minutes of their owning app being moved to the background. We observed that before these services are killed, they are still able to launch activities from the background without any issues despite Android Q's effort to block this action. It is worth noting that this vulnerability is not readily usable in our attack as we need our attack app to indefinitely run in the background. However, we observed another way that we can achieve similar results to our original attack. More specifically, we discovered that activity starts from within other activities are not checked by the system as a part of this new security mechanism. Here, the possible assumption made by the system is that activities will be stopped automatically (i.e., they will not run any code) when they move to the background; hence, there is no need to enforce the background activity start protocol on activities as they cannot possibly achieve this. However, we show this is a false assumption since activities that are moved to the background via the `moveTaskToBack` API are still kept running by the system. This, in turn, gives a background activity the ability to start new activities without getting blocked on Android Q. Activities started in this way are also automatically moved to the foreground by the system, as we have observed. We utilize this finding in our attacks on Android Q; instead of having a `JobScheduler` service, we create a background activity to carry out the tasks explained in Section 6.3.3. This activity can be launched when the user presses on the back button to exit the attack app or simply when the user switches to another app. Up until this point, the attack app can simply pretend to provide its deceivingly-useful functionality (e.g., QR code scanning etc.). However, since the attacking component of the app is now an activity and not a service, the attack app will appear in the recents screens even when it is on the background (when the recents button is clicked by the user); hence, there is a possibility that the app will be seen by the user at some point in time and get killed or even uninstalled. In order to evade detection, we ensure that the user will not notice that the attack app is running on the background by removing the app from the task stack so that it will not be shown in the recents screen. We achieve this by setting the `android:excludeFromRecents` flag of the activities of the app in the manifest file, as described in Section 2.1.4. Additionally, in order to handle cases where the attack app gets killed for any other reason, we can utilize a sticky service that relaunches the attack activity and immediately moves it to the background to continue running our attack indefinitely.

6.7 DISCUSSION

Here we discuss some of the concerns that might be raised by the reader, in a Q&A format.

• *Why doesn't the attacker just launch a normal phishing attack?* Google Play (GP) store already has some security mechanisms in place to detect phishing attacks. For example, GP does not allow apps to be published with icons that are similar to those of other apps. In addition, GP can also identify if the title of an app is dangerously similar to that of another app. If an app has some suspicious behavior as in these cases, it will be suspended by GP indefinitely. With our attack, the adversary does not run into the problem of getting detected by Google Play store.

In addition, users are generally familiar with the concept of classic phishing attacks, where the attacker impersonates another app by mimicking its UI. Hence, it is more likely (compared to our attacks) that they will be on the lookout for such attacks. Our attacks do not require mimicking another app's UI and are previously unknown to the users (as well as to the research and developer communities). Therefore, users will be highly vulnerable and will get caught off-guard as they are not expecting such attacks in the first place. We have proven the validity of this statement with our user study.

• *Couldn't the app just pretend to do something useful with the permission to convince the users to grant it?* While it is true that the most important reason for users to grant permissions is the permission's relevance to utility, it is not the sole factor that plays a role in these decisions. Previous work has shown that users consider the relevance of the permission to the app's functionality (utility) for 68% of all their grant decisions, and for 32% of all grant events, they consider the reputation of the app developer as an important factor influencing them to grant a permission [48]. In total, users select both criteria for 21% of all the times they decided to grant permissions[2]. This means that only for 47% of all the grant events, users would agree to grant permissions to a non-reputable app developer even if the app could make a strong case of needing a permission. Since 86% of all permission decisions (both grant and deny) corresponds to grant events as reported in [48], in total we have a final 40% grant rate for non-reputable apps for all permission events. As can be seen, this rate is much lower than the promised success rate of our attack (above 80%).

In general, both our user study and previous studies show that users do try to make conscious decisions when it comes to permissions. They feel more comfortable granting permissions to some apps while they do not feel so for others based on several factors. Our goal is to enable adversaries to take advantage of user's trust in another app, without having to gain that trust on their own.

• *What is the attacker going to do with the permissions?* Our attacks serve as a platform for different adversaries to obtain any desired permissions to realize their own goals. Each adversary can come up with a different attack strategy that requires them to obtain a specific set of permissions, which they could achieve using our attack scheme.

[2]This number is not reported in the cited article [48], we contacted the authors to obtain this information.

6.8 SUMMARY

In this chapter, we presented a new class of phishing-based privilege escalation attacks on Android runtime permissions. We conducted user studies to understand the effects of users' knowledge of runtime permissions and their past behavior in this context on the success of our attacks. By doing this, we identified the most favorable circumstances for our attacks and utilized these findings to make our attacks more realistic. At the very least, our user study shows that our attacks are plausibly effective against many users, as all of our participants were oblivious to the changes induced by the attack and followed through with the attack without noticing anything suspicious. In addition, we discovered vulnerabilities in Android's most recent mitigation approach against phishing attacks implemented on the latest Android release (10.0 Q), which allowed us to still be able to launch our attacks even on this "phishing-free" Android version without any problem.

CHAPTER 7: CONCLUSION

7.1 SUMMARY

In this thesis, we conduct extensive analysis on issues that cause over-privilege in mobile cross-origin interactions. We choose the Android operating system as a use case due to its popularity and open-source nature and show how such issues constitute a significant impediment to establishing least privilege on this platform. More specifically, we demonstrate how adversaries can escalate their privileges to obtain unfettered access to sensitive resources, threatening the security of millions of apps and billions of mobile device users. As a remedy, we provide practical and systematic access control solutions to realize least privilege on mobile platforms. In particular, we investigate the security of the interactions between apps and 1) web domains, 2) other apps, and 3) the system. First, we identify the lack of access control for the interactions between apps and web domains for all apps that utilize embedded web browsers and introduce a fine-grained and uniform access control mechanism to allow developers to specify policy rules on which resources they want to expose to certain web domains. Then, we investigate the security of the main access control mechanism for IPC on Android, namely custom permissions, and identify serious security vulnerabilities in this mechanism that lead to privilege escalation attacks on high-risk system resources, sensitive user data, and private app resources. We introduce a new design to custom permissions to systematically address these vulnerabilities by targeting their root cause and we conduct a formal analysis on our approach to prove its correctness. Finally, we investigate the security of Android's new runtime permission model.We show that this model fails to meet its key security guarantees as we discovered ways for adversaries to break the implicit assumptions the model relies on for its correct operation to launch phishing-based privilege escalation attacks on high-risk platform resources and sensitive user data. We conduct user studies to help us build realistic attacks. We also investigate Android's current approach to completely eliminating phishing on the platform and show how this approach fails to achieve this goal due to existing security vulnerabilities we discovered, rending our attacks still viable.

All in all, in this thesis we identified issues that threaten least privilege on mobile platforms and provided practical mitigation solutions to resolve such issues in a systematic, effective, and efficient manner. Our work has influenced the design and implementation of some of the critical security mechanisms in Android and have led to changes in the official Android releases. However, the vulnerabilities and design issues we disclosed in this thesis are not confined only to the Android platform and can in fact be found in other mobile operating systems that utilize cross-origin interactions. In addition, our methodologies and mitigation techniques serve as general-purpose solutions that can be adapted to other platforms.

7.2 HIGHER-LAYER MITIGATION STRATEGIES

End-to-end principle in system design states that placing functions at low layers of a system might mean redundant effort or provide little value; hence, implementing such functions at higher levels of the system can prove to be more efficient or reliable. Given this principle, one could argue that higher-layer solutions could be just as effective for the issues we unearthed in this work with the additional benefit of easier implementation, error handling, and deployment while avoiding redundant efforts. For example, with the custom permission vulnerabilities of Chapter 5 and the phishing-based privilege escalation attacks on runtime permissions of Chapter 6, we have shown that adversaries can spoof custom permission names and internal app names respectively to wreak havoc on the platform. In addition, app repacking attacks [97] and mobile deep-link spoofing attacks [98, 99] have also been known to similarly target package names, and mobile deep links, respectively. A majority of these issues have been addressed by issuing mitigation strategies in different parts of the platform as of now. However, the existence of all these attacks that rely on a similar kind of exploit on different types of namespaces point out to a deeper problem we would like to summarize here as the *platform's implicit trust in developer-defined namespaces of apps.* Given how the root cause of all these issues can be effortlessly expressed under one umbrella, it might be reasonable to attempt to tackle them collectively at a higher level where they can be more easily addressed while avoiding redundant mitigation efforts. What comes to mind as a sensible approach in this regard is to implement a security mechanism within app markets such as Google Play and Apple's app store, to detect any kind of spoofing on *all* of the developer-defined namespaces and suspend suspicious apps that are involved in such behavior. As Apple's app store is the only source where users can download iOS apps from, this approach would work on the iOS platform. However, on Android, even though Google Play store is the official app store, apps can still be downloaded from unofficial app markets or they can be sideloaded. This makes it virtually impossible to adopt such a higher-level strategy on the Android platform. Please note that we are not claiming a higher-level approach is absolutely the right strategy here, we are only considering such a solution as a possibility given the end-to-end argument.

7.3 FUTURE DIRECTIONS

Moving forward, other channels for cross-origin interactions on mobile platforms could be investigated in the future for security and privacy. Some examples include the interactions of apps with system services and mobile deep linking channels [98] on the platform. In addition, we envision that off-platform cross-origin interactions with other critical systems, such as the Internet-of-Things devices and smart automobiles, create significant avenues for future work as these systems are conjectured to become mainstream in our lives and are expected to heavily interact with smartphones and similar devices that run mobile operating systems.

APPENDIX A: CUSTOM PERMISSIONS

A.1 IMPLEMENTATION OF CUSPER IN ALLOY

In our formal model of Android runtime permissions, we update the representation of permissions in order to reflect our design decisions. First, we add a boolean field to our `Permission` Alloy signature to indicate whether a permission is custom or system. Then, we also add a source id field, which will be used during permission enforcement to uniquely identify permissions. Here, we use app signature and not the hash of it for our formal model for simplicity. Updated permission abstraction can be seen in Listing A.1.

For app and component guards, Cusper performs name translation at runtime during enforcement. Listing A.2 represents the component invocation operation and the predicate in line 8 illustrates how permission enforcement is done according to Cusper. Whenever a component is being invoked, we retrieve the name of corresponding app or component permission, perform a lookup operation to find the corresponding Cusper name (see line 11). In addition, we update how we perform enforcement, so that when the system checks whether a calling app has the permission required to invoke a component, it will use both the permission name and the source id (see line 14).

Furthermore, we utilize the boolean custom permission indicator field to correctly set the `FLAG_PERMISSION_REVOKE_ON_UPGRADE` flag as shown in line 25 of Listing A.3, which illustrates how `grantInstall` case is handled.

A.2 OTHER DISCOVERED ATTACKS

In addition to the attacks we discussed in Section 5.3, we also discovered another attack on Android custom permissions that utilizes the lack of naming conventions for permissions

Listing A.1: New permission representation according to Cusper

```
sig Permission {
  name: PermName,
  protectionLevel: ProtectionLevel,
  sourcePackage :PackageName,
  isCustomPermission: Bool, // new field for Cusper
  permGroup: lone PermGroupName,
  sourceId: AppSignature // new field for Cusper
}
```

to launch attacks on benign apps [100].

As described in Section 2.2.3, custom permissions can also be created dynamically via the Android APIs. In this attack, the adversary spoofs the dynamic custom permissions of the victim. This attack is currently reproducible only on older versions of (<6) due to some other issues in the new versions. Since we focused on modeling the new versions of Android and did not find strong evidence for the use of dynamic permissions by third-party developers as of now, we did not address this attack in our work. However, we believe it is worth presenting here as it demonstrates the extent of custom permission vulnerabilities and provides further proof that Android custom permissions are problematic at their current stage.

Steps to Produce the Attack. In order to carry out the dynamic custom permission attack, the adversary builds an app that statically declares a custom permission that the victim app is planning to dynamically create via the `addPermission()` API method, which requires the static declaration of a permission tree with a specific domain name by the victim. The attack can work only if the installation of the attack app is performed before the victim app has an opportunity to dynamically create the permission of interest. After this, the attack app can gain unfettered access to signature protected components of the victim app, while the victim will not be able to dynamically create its own custom permission anymore since it is already defined in the system.

Internals of the Attack. Android does not seem to perform any checks on the availability of the permission names for statically defined custom permissions against the permission tree names on the device. In other words, an app can still statically declare a custom permission with the domain name of a permission tree declared by another app, even though the operation would fail if the app tried to declare this permission dynamically (i.e., throws `SecurityException` stating the tree belongs to another app). Same kind of name translation approach we presented in Cusper can be used for the names of permission trees to resolve this problem.

As we mentioned, this attack works only on older Android versions (<6.0) since the new versions require `MANAGE_USERS` or `CREATE_USERS` permissions for the `addPermission()` API to properly work even though this behavior is not documented in the Android developer guides. We believe this itself might be an undesired behavior that was introduced by the system developers while implementing possibly the multi-user framework in Android; hence, if this implementation is changed, the dynamic custom permission spoofing vulnerability should emerge on Android 6.0 and onward.

Listing A.2: Component invocation with Cusper

```
pred invoke[t, t' :Time, caller, callee: Component]{
   caller.app + callee.app in Device.apps.t
   canCall[caller, callee, t]
   noChanges[t, t']
}

// Permission enforcement in the model according to Cusper
pred canCall[caller, callee: Component, t :Time] {
   let pname = guardedBy[callee],
      // name translation during enforcement for components
      source = getSourceId[callee, t],
      pd = getPermissionData[pname, caller.app, t] {
         pname in pd.perm.name
         source in pd.perm.sourceId
      }
}

// Return name of the permission protecting component
fun guardedBy :Component -> PermName {
   {c: Component, p: Name |
      // component-specific permission takes priority over the app-wide
          permission
      (p = c.guard.name) or (no c.guard and p = c.app.guard.name)
   }
}

// Guard names are translated during enforcement
fun getSourceId[c: Component, t: Time] :one AppSignature {
   {p: AppSignature |
      (p = findPermissionByName[c.guard.name, t].sourceId) or
      (no c.guard and p = findPermissionByName[c.app.guard.name, t].sourceId)
   }
}

// No changes in device and granted permissions
pred noChanges[t, t': Time] {
   Device.customPerms.t' = Device.customPerms.t
   Device.apps.t' = Device.apps.t
   all a :Application | a.permissionsState.t' = a.permissionsState.t
}
```

Listing A.3: Grant install case for grantPermissions predicate

```
pred grantInstallCase[p: Permission, app : Application, t, t' : Time] {
   hasRuntimePermission[p, app, t]
      ⟹ revokeRuntimePermission[p, app, t, t']
         grantInstallPermission[p, app, t, t']
   // no runtime permission should exist for this permission
   no pd: app.permissionsState.t' |
      pd.perm = p and pd.isRuntime = True
}

pred hasRuntimePermission[p: Permission, app : Application, t : Time] {
   one pd: app.permissionsState.t |
      pd.perm.name = p.name and pd.isRuntime = True
}

pred revokeRuntimePermission[p: Permission, app : Application, t, t' : Time] {
   no pd: app.permissionsState.t' |
      pd.perm.name = p.name and pd.isRuntime = True
}

pred grantInstallPermission[p: Permission, app : Application, t, t' : Time] {
   one pd: app.permissionsState.t' |
      pd.perm = p and pd.isRuntime = False
      // Cusper fix: clear the flag to disallow automatic upgrade to runtime
          for custom permissions
      p.isCustomPermission = False ⟹clearPermFlags[pd, t']
      else setPermFlags[pd, t']
}

pred clearPermFlags[pd: PermissionData, t: Time] {
   pd.flags.FLAG_PERMISSION_REVOKE_ON_UPGRADE = False
   // clear other flags...
}

pred setPermFlags[pd: PermissionData, t: Time] {
   pd.flags.FLAG_PERMISSION_REVOKE_ON_UPGRADE = True
   // set other flags...
}
```

APPENDIX B: RESEARCH ETHICS AND SAFETY

We obtained approval from the Institutional Review Board (IRB) at the University of Illinois at Urbana-Champaign for our study described in Chapter 6 prior to the commencement of the research. This approval was received under the protocol title "Improving Android Runtime Permissions". Our user study comprised of an online survey where we recruited participants from Amazon Mechanical Turk (mTurk). Prior to our study, we received consent from all of our participants to collect and share their data anonymously.

REFERENCES

[1] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *CCS*, 2013.

[2] Android open source project. `https://source.android.com/`.

[3] Güliz Seray Tuncay, Soteris Demetriou, and Carl A. Gunter. Draco: Uniform and fine-grained control of web code access on android. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2016.

[4] National vulnerability database. CVE-2017-0593 detail. `https://nvd.nist.gov/vuln/detail/CVE-2017-0593`.

[5] National vulnerability database. CVE-2019-2200 detail. `https://nvd.nist.gov/vuln/detail/CVE-2019-2200`.

[6] Güliz Seray Tuncay, Soteris Demetriou, Karan Ganju, and Carl A. Gunter. Resolving the predicament of android custom permissions. In *Proceedings of the 2018 ISOC Network and Distributed System Security Symposium (NDSS)*, 2018.

[7] Don't allow permission change to runtime. aosp bug fix. `https://android.googlesource.com/platform/frameworks/base/+/78efbc95412b8efa9a44d573f5767ae927927d48`.

[8] Android security acknowledgements. `https://source.android.com/security/overview/acknowledgements`.

[9] Insights into the 2.3 billion android smartphones in use around the world. `https://newzoo.com/insights/articles/insights-into-the-2-3-billion-android-smartphones-in-use-around-the-world/`.

[10] Android version history. `https://en.wikipedia.org/wiki/Android_version_history`.

[11] Android platform architecture. `https://developer.android.com/guide/platform`.

[12] Understand the activity lifecycle. `https://developer.android.com/guide/components/activities/activity-lifecycle`.

[13] Openbinder. `http://www.angryredplanet.com/~hackbod/openbinder/docs/html/`.

[14] Webkit: Open source web browser engine. `https://webkit.org/`.

[15] The chromium project. `https://chromium.org/`.

[16] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android permissions remystified: A field study on contextual integrity. In *USENIX Security*, 2015.

[17] App permissions. `https://developer.android.com/guide/topics/permissions/overview`.

[18] Android permissions. `https://tinyurl.com/y863owbb`.

[19] Request app permissions. `https://developer.android.com/training/permissions/requesting`.

[20] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM, 2014.

[21] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[22] Androguard. `https://github.com/androguard/androguard`.

[23] Alloy: A language and tool for relational models. `http://alloy.mit.edu`.

[24] D. McCracken and E. Reilly. Backus-naur form (bnf). In *Encyclopedia of Computer Science*.

[25] S. Shekhar, M. Dietz, and D. Wallach. Adsplit: Separating smartphone advertising from applications. In *USENIX*, 2012.

[26] J. Seo, D Kim, D Cho, T. Kim, and I. Shin. Flexdroid: Enforcing in-app privilege separation in android. 2016.

[27] S. Zhu, L. Lu, and K. Singh. Case: Comprehensive application security enforcement on cots mobile devices. In *MobiSys*, 2016.

[28] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the android system. In *ACSAC*. ACM, 2011.

[29] E. Chin and D. Wagner. Bifocals: Analyzing webview vulnerabilities in android applications. In *Information Security Applications*. 2013.

[30] P. Mutchler, A. Doupé, J. Mitchell, C. Kruegel, and G. Vigna. A large-scale study of mobile web app security. In *MoST*, 2015.

[31] M. Neugschwandtner, M. Lindorfer, and C. Platzer. A view to a kill: Webview exploitation. In *LEET*, 2013.

[32] D. Thomas, A. Beresford, T.s Coudray, T. Sutcliffe, and A. Taylor. The lifetime of android api vulnerabilities: case study on the javascript-to-java interface. In *Security Protocols XXIII*. 2015.

[33] Y. Jing and T. Yamauchi. Access control to prevent malicious javascript code exploiting vulnerabilities of webview in android os. *IEICE TRANSACTIONS on Information and Systems*, 2015.

[34] M. Georgiev, S. Jana, and V. Shmatikov. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *NDSS*, 2014.

[35] X. Jin, L. Wang, T. Luo, and W. Du. Fine-grained access control for html5-based mobile applications in android. In *Information Security*. 2015.

[36] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *CCS*, 2014.

[37] M. Georgiev, S. Jana, and V. Shmatikov. Rethinking security of web-based system applications. In *WWW*, 2015.

[38] A.r Felt, Helen J Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: and defenses. In *USENIX Security*, 2011.

[39] E. Chin, A. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *MobiSys*, 2011.

[40] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Permission evolution in the android ecosystem. In *ACSAC*, 2012.

[41] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *CCS*, 2011.

[42] H. Bagheri, E. Kang, S. Malek, and D. Jackson. Detection of design flaws in the android permission protocol through bounded verification. In *International Symposium on Formal Methods*, 2015.

[43] G. Betarte, J. Campo, C. Luna, and A. Romano. Verifying android's permission model. In *Theoretical Aspects of Computing*, 2015.

[44] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka. A formal model to analyze the permission authorization and enforcement in the android framework. In *SocialCom*, 2010.

[45] Y. Zhauniarovich and O. Gadyatskaya. Small changes, big changes: an updated view on the android permission system. In *RAID*, 2016.

[46] Z. Fang, W. Han, D. Li, Z. Guo, D. Guo, X. Wang, Z.n Qian, and H. Chen. revdroid: code analysis of the side effects after dynamic permission revocation of android apps. In *Asia CCS*, 2016.

[47] D. Bogdanas, N. Nelson, and D. Dig. Analysis and transformations in support of android privacy. Technical report, 2016.

[48] Bram Bonné, Sai Teja Peddinti, Igor Bilogrevic, and Nina Taft. Exploring decision making with android's runtime permission dialogs using in-context surveys. In *Thirteenth Symposium on Usable Privacy and Security ({SOUPS} 2017)*, pages 195–210, 2017.

[49] Custom permission vulnerabilities. `https://tinyurl.com/y7yoae52`.

[50] W. Shin, S. Kwak, S. Kiyomoto, K. Fukushima, and T. Tanaka. A small but non-negligible flaw in the android permission scheme. In *POLICY*, 2010.

[51] J. Sellwood and J. Crampton. Sleeping android: The danger of dormant permissions. In *SPSM*, 2013.

[52] Adrienne Porter Felt and David Wagner. *Phishing on mobile devices.* 2011.

[53] Simone Aonzo, Alessio Merlo, Giulio Tavella, and Yanick Fratantonio. Phishing attacks on modern android. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1788–1801. ACM, 2018.

[54] Qi Alfred Chen, Zhiyun Qian, and Z Morley Mao. Peeking into your app without actually seeing it:{UI} state inference and novel android attacks. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 1037–1052, 2014.

[55] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. No pardon for the interruption: New inference attacks on android through interrupt timing analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 414–432. IEEE, 2016.

[56] Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. Procharvester: Fully automated analysis of procfs side-channel leaks on android. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 749–763. ACM, 2018.

[57] Raphael Spreitzer, Gerald Palfinger, and Stefan Mangard. Scandroid: Automated side-channel analysis of android apis. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 224–235. ACM, 2018.

[58] Marcus Niemietz and Jörg Schwenk. Ui redressing attacks on android devices. *Black Hat Abu Dhabi*, 2012.

[59] Longfei Wu, Benjamin Brandt, Xiaojiang Du, and Bo Ji. Analysis of clickjacking attacks and an effective defense scheme for android devices. In *2016 IEEE Conference on Communications and Network Security (CNS)*, pages 55–63. IEEE, 2016.

[60] Yanick Fratantonio, Chenxiong Qian, Simon P Chung, and Wenke Lee. Cloak and dagger: from two permissions to complete control of the ui feedback loop. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1041–1057. IEEE, 2017.

[61] Andrea Possemato, Andrea Lanzi, Simon Pak Ho Chung, Wenke Lee, and Yanick Fratantonio. Clickshield: Are you hiding something? towards eradicating clickjacking on android. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1120–1136. ACM, 2018.

[62] Dex2jar. `https://github.com/pxb1988/dex2jar`.

[63] JD-Gui. `http://jd.benow.ca/`.

[64] Apktool decompiler. `http://ibotpeaches.github.io/Apktool/`.

[65] Eddystone BLE beacons. `http://bit.ly/1WMaylQ`.

[66] Bluetooth low energy. `http://bit.ly/1Rw9grs`.

[67] 15 companies using beacon technology. `http://bit.ly/16qwASy`.

[68] The apktool's failed app list. `http://bit.ly/2aUyE9T`.

[69] K. Au, Y. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *CCS*, 2012.

[70] DexDump. `http://bit.ly/1NBg7QM`.

[71] Cold start times: Analysis of top apps. `http://bit.ly/1TFTtb0`.

[72] Key takeaways for mobile apps. `http://pewrsr.ch/1M4LqyY`.

[73] Android dashboard. `https://tinyurl.com/qfquw3s`.

[74] Privilege escalation through custom permission update. `https://issuetracker.google.com/issues/37130844`.

[75] Android : Requesting permissions. `https://tinyurl.com/y8gp4dn6`.

[76] Google cloud messaging. `https://tinyurl.com/ybocrrqw`.

[77] Upload applications to appaloosa. `https://tinyurl.com/y94pb3cv`.

[78] Privilege escalation by exploiting fcfs property of custom permissions. `https://issuetracker.google.com/issues/37131935`.

[79] Creating apps with plugin architecture. `https://tinyurl.com/ydfdk9z7`.

[80] Android plugin application. `https://tinyurl.com/ycfd9pot`.

[81] Yoga guru. `https://tinyurl.com/yb3dqopp`.

[82] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang. Upgrading your android, elevating my malware: Privilege escalation through mobile os updating. In *IEEE Security and Privacy*, 2014.

[83] Program correctness, the specification. `https://tinyurl.com/y8r8cze8`.

[84] K. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming AI and Reasoning*, 2010.

[85] Resolving the predicament of android custom permissions. `https://sites.google.com/view/cusper-custom-permissions/home`.

[86] Android users have an average of 95 apps installed on their phones, according to yahoo aviate data. `https://tinyurl.com/ybc7dqbn`.

[87] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2012.

[88] Dongjing He, Muhammad Naveed, Carl A Gunter, and Klara Nahrstedt. Security concerns in android mhealth apps. In *AMIA Annual Symposium Proceedings*, volume 2014, page 645. American Medical Informatics Association, 2014.

[89] Piotr Sapiezynski, Arkadiusz Stopczynski, Radu Gatej, and Sune Lehmann. Tracking human mobility using wifi signals. *PloS one*, 10(7):e0130824, 2015.

[90] Sashank Narain, Triet D Vo-Huu, Kenneth Block, and Guevara Noubir. Inferring user routes and locations using zero-permission mobile sensors. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 397–413. IEEE, 2016.

[91] Yan Michalevsky, Aaron Schulman, Gunaa Arumugam Veerapandian, Dan Boneh, and Gabi Nakibly. Powerspy: Location tracking using mobile device power analysis. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 785–800, 2015.

[92] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. What the app is that? deception and countermeasures in the android user interface. In *2015 IEEE Symposium on Security and Privacy*, pages 931–948. IEEE, 2015.

[93] Background execution limits. `https://developer.android.com/about/versions/oreo/background`.

[94] Luis Leiva, Matthias Böhmer, Sven Gehring, and Antonio Krüger. Back to the app: the costs of mobile application interruptions. In *Proceedings of the 14th international conference on Human-computer interaction with mobile devices and services*, pages 291–294. ACM, 2012.

[95] Android device names. `https://github.com/jaredrummler/AndroidDeviceNames`.

[96] Android Q privacy change: Restrictions to background activity starts. `https://developer.android.com/preview/privacy/background-activity-starts#display-notification-user`.

[97] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*, pages 95–109. IEEE, 2012.

[98] Fang Liu, Chun Wang, Andres Pico, Danfeng Yao, and Gang Wang. Measuring the insecurity of mobile deep links of android. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 953–969, 2017.

[99] Luyi Xing, Xiaolong Bai, Tongxin Li, XiaoFeng Wang, Kai Chen, Xiaojing Liao, Shi-Min Hu, and Xinhui Han. Cracking app isolation on apple: Unauthorized cross-app resource access on mac os. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 31–43. ACM, 2015.

[100] Privilege escalation by exploiting permission trees and dynamic custom permissions. `https://issuetracker.google.com/issues/37324008`.