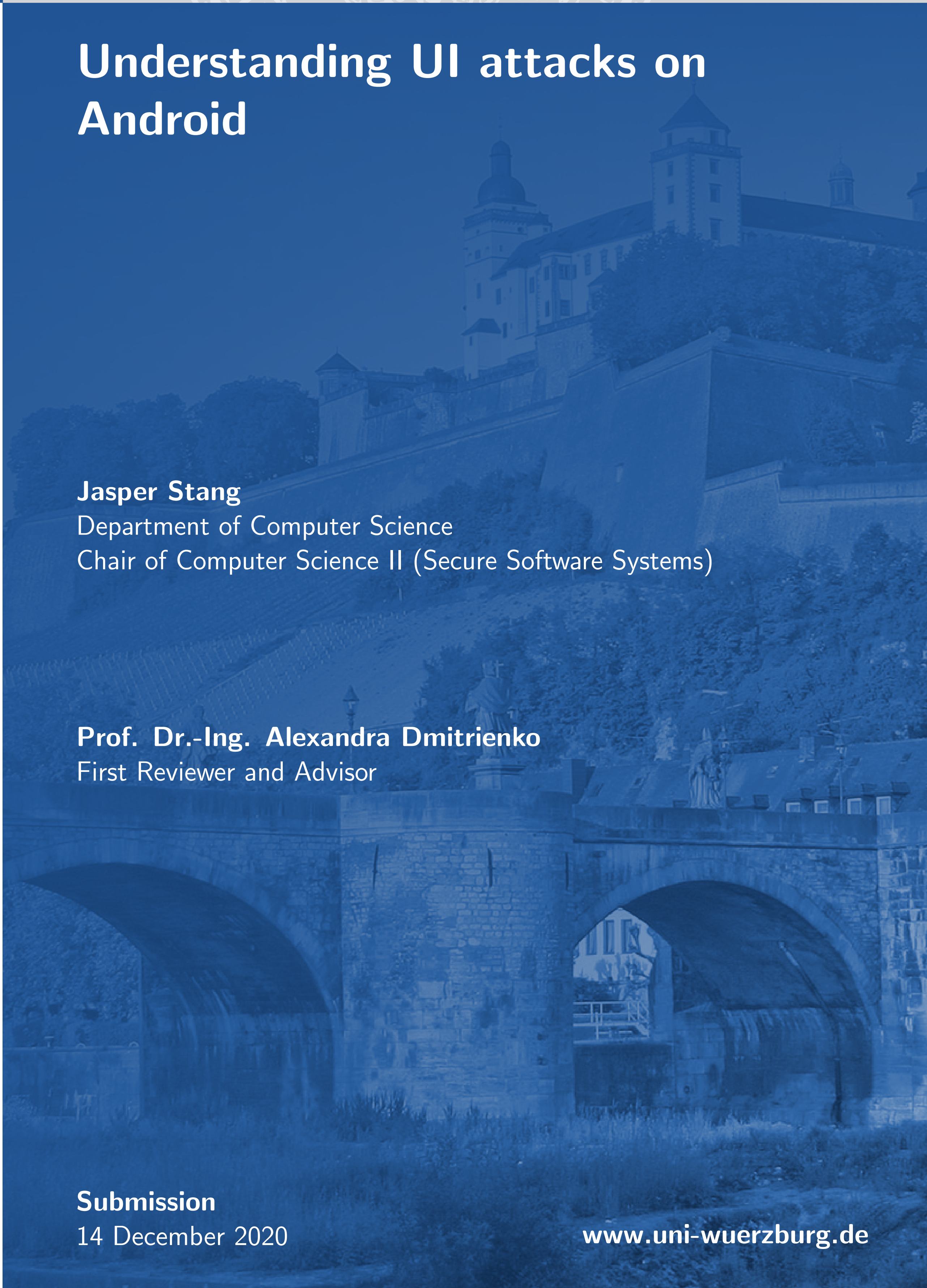
Bachelor Thesis



Understanding UI attacks on Android

Jasper Stang
Department of Computer Science
Chair of Computer Science II (Secure Software Systems)

Prof. Dr.-Ing. Alexandra Dmitrienko
First Reviewer and Advisor

Submission
14 December 2020

www.uni-wuerzburg.de

Abstract

In this thesis, we closely examine an important type of attack against Android smartphones that exploit weaknesses in the user interface. In particular, we study, analyze and implement the "Keystroke Inference #3" attack from the "Cloak and Dagger" paper by Fratantonio et al. [1, 2], which enables an attacker to steal sensitive input such as passwords. The attack takes advantage of a vulnerability that was subsequently patched on all newer Android versions. Yet, it still affects a significant user base that utilizes older devices. In this paper, we present the end-to-end attack implementation of the "Keystroke Inference #3" concept and elaborate on in-depth details. In order to make the attack feasible certain technical challenges needed to be solved, therefore our developed approaches are presented as well. After the evaluation of the results, we show that the implementation is applicable to a wide range of Android versions. We then present our novel defense technique `OverlayShifter`, which fully prevents the attack while being independent of operating system modifications. Moreover, characteristics that facilitate the detection of the attack are discussed.

Zusammenfassung

In dieser Arbeit beschäftigen wir uns mit User-Interface-Angriffen gegen Android Smartphones. Diese stellen einen besonders wichtigen Angriffsvektoren dar. Vor allem untersuchen und implementieren wir den "Keystroke Inference #3"-Angriff, der schematisch von Fratantonio et al. im Jahr 2017 vorgestellt wurde [1, 2]. Dieser Angriff erlaubt es die Eingaben von Nutzern mitzuverfolgen und somit unter anderem Passwörter oder sonstige sensible Daten zu erbeuten. Zwar wird bei dem Angriff eine Sicherheitslücke genutzt, die bereits vor einiger Zeit geschlossen wurde, jedoch wird ein signifikanter Teil der Nutzer trotzdem nicht vor solchen Angriffen geschützt, da diese veraltete Mobiltelefone benutzen und somit keinen Zugriff mehr auf Sicherheitsupdates haben. Zunächst präsentieren wir unsere Implementierung für den Angriff. Um den Angriff erfolgreich durchführen zu können mussten einige technische Schwierigkeiten überwunden werden, deshalb erläutern wir zusätzlich unsere entwickelten Ansätze um diese zu lösen. Im weiteren Verlauf der Arbeit präsentieren wir die Auswertung des implementierten Angriffs. Hierbei zeigen wir, dass der Angriff bei einer Vielzahl an Geräten durchführbar ist. Zudem präsentieren wir unsere neuartige Verteidigungsmethod `OverlayShifter`. Diese ist in der Lage eine erfolgreiche Durchführung des Angriffs, ohne Betriebssystemmodifikationen, zu verhindern. Außerdem stellen wir Eigenheiten vor, die einer Erkennung von User-Interface-Angriffen dienen.

Contents

1. Introduction

Mobile phones are playing an extremely important role in our everyday life. A majority of these devices use the Android operating system [3] and are used for highly security-sensitive tasks like online banking, online shopping and other services concerning financial transactions. It is therefore inevitable that criminals develop malware that exploits vulnerabilities on smartphones to enrich themselves. Several attacks have already been discovered, such as the "HummingBad" malware, which infected an estimated ten million devices and deployed ads to generate revenue [4]. Another example is the "ZeuS-in-the-Mobile" malware, which aimed at stealing mobile TANs used to authorize transactions in online banking [5]. By using sophisticated exploitation techniques, Davi et al. [6] managed to bypass restrictions imposed by the sandbox and achieved privilege escalation. This can, for instance, be abused to send text messages to any phone number without the user's knowledge. Currently, computer scientists are putting a lot of effort into developing innovative techniques to detect Android malware, e.g. [7, 8, 9].

The ubiquitous digitization and rapid advancement of technology not only lead to a better user experience but also made evident the importance of security and data protection. Criminals have been trying to steal sensitive information for decades and their attempts are becoming increasingly more frequent and sophisticated over time. As a result, almost every device is exposed to threats. The User Interface (UI) attacks form an important attack vector. For years, researchers have been stressing that the Android user interface (cf. [10], [11]) should be considered a security-sensitive part of the software. There are many examples of poor design choices of the UI that lead to security vulnerabilities, for instance, Clickjacking [10] or general UI deception [11]. When exploited, these vulnerabilities can deceive the user into unintentionally giving access to their sensitive information without their knowledge. In 2017, Fratantonio et al. [1] demonstrated in their paper "Cloak and Dagger" how a critical flaw in Android's UI can lead to a complete bypass of the security measures.

Developing defenses against this type of attacks is challenging. Proposed defense mechanisms are either based on heuristic approaches, which may suffer from false positives and are therefore only partially suitable for the average user [11], or on hybrid approaches that combine dynamic and static analysis [12]. It is proven that these are still not able to detect all UI attacks and suffer to some extent from race conditions [11]. Furthermore, some of them have to be applied in pre-deployment [13] and are therefore dependent on being implemented by app stores. In case the malware is installed from sources other than app stores the defense mechanism will not be applied. Moreover, defense methods directly targeting the attack have a significant negative impact on the usability of a device or are not able to fully prevent the attack.

This work focuses on the "Keystroke Inference #3" attack from the "Cloak and Dagger" paper [1], which is also called "Invisible Grid" attack [2]. By means of this attack, an adversary is able to eavesdrop on sensitive input by utilizing invisible overlay windows. The attack itself was subsequently patched on almost all Android versions, however, a significant user base utilizes older devices that may not receive security fixes anymore. Hence, defense methods which are independent of Operating System (OS) modification need to be applied in order to prevent the attack. Fratantonio et al. [1] identified the side-channel enabling the attack, however, they did not present a proof-of-concept implementation. Important technical challenges that are essential for an end-to-end attack implementation such as keyboard detection and layout retrieval remain unsolved. Hence, in this work, the operating principle of this attack will be investigated in-depth and solutions for the technical challenges will be provided. In particular, this thesis makes the following contributions:

- We investigate the operating principle of this attack.
- Furthermore, important technical challenges such as system-wide keyboard presence detection, mapping of clicks to specific keyboard keys and keyboard type detection are solved.
- The end-to-end attack implementation is evaluated and the development of further countermeasures is facilitated by providing attack-specific characteristics.
- We present our, system-wide working, novel defense technique called `OverlayShifter` which fully prevents the "Keystroke Inference #3" attack while preserving overlay functionality, without being dependent on OS modification.

Furthermore, the execution of this malware sample can be used to obtain an attack trace that can be collected. These traces can later be used in projects that use e.g. machine learning techniques to detect UI attacks. This detection technique should then be applied as an alternative to a security fix, which may not be available for older devices.

The main objective of this thesis is to build the foundation for future research. The attack is therefore implemented for Android 7.0 Versions that have not patched the side-channel exploited by "Keystroke Inference #3" attack. Furthermore, the execution of the end-to-end attack sample can be used to produce an attack trace, which is used for future analysis or evaluation of detection methods. In addition to these objectives, we discuss attack-specific characteristics that indicate an ongoing attack. The results can be used to further facilitate the development of countermeasures. Moreover, we present our novel defense technique `OverlayShifter` which is able to fully prevent the attack without being dependent on OS modification.

Sections of the Thesis This work is structured as follows: In Chapter 2, we present background information about the Android operating system. User interface attacks and the "Invisible Grid" [2] attack in particular are introduced in Chapter 3. We present the problem statement in Chapter 4 followed by our identification of essential requirements and technical challenges for the attack implementation in Chapter 5. The solutions to identified technical challenges and the end-to-end attack implementation are then elaborated in-depth in Chapter 6. We evaluate the attack implementation in Chapter 7. We present our novel defense technique in Chapter 8. Additionally, we propose further defense mechanisms and present characteristics of the "Invisible Grid" attack in Chapter 9. Finally, conclusions are drawn in Chapter 10.

2. Background

In this chapter, we present background information about the Android operating system to comprehend the way the attack is operating.

2.1. Android Security Mechanisms

Key security mechanisms of the Android OS are the advanced permission model, process isolation and application sandboxing. Internally, an Android app is associated with a Linux process. The Linux operating system enforces security by using a kernel-level sandbox for each new process. This sandbox mechanism is achieved by assigning each process a unique user ID. This prevents processes from communicating with each other and limits the access to the operating system [14]. Additionally, a new so-called *Dalvik* Virtual Machine (VM) is spawned inside each process in order to support the execution of Java-based applications. The VM then loads the pre-compiled Java code and executes it. The Android operating system provides an easy-to-use Application Programming Interface (API) to access all device-related hardware, e.g., the internal memory, GPS or WI-FI. It also provides access to software methods, such as interfaces for spawning new windows or setting the background image.

Such API functions may utilize underlying system resources or require access to security-sensitive functionality or private data. It is, therefore, useful to provide some kind of restriction. To enforce these restrictions, Android introduced a permission model. The permissions are divided into four different levels. The first level also referred to as install-time permissions, is the group in which API functions pose little potential risk to the user or the system. Those permissions are granted by the system without consent of the user. The second level, which is called normal permissions, contains all API calls that have a high potential to be abused and are therefore protected stronger. Those API calls, for instance, may access data on the phone. Those permissions can only be granted after the user explicitly gave his consent. The third group is called signature permissions. Those are highly dangerous and therefore only granted to applications that are signed with the same certificate as the application that defined this permission. The last level is called runtime permissions all API calls that directly access privacy-sensitive user data such as the location or the contacts fall into this category. Those permissions must therefore be granted by the user during runtime of the app [15].

As the operating system gives permissions to Linux processes, restrictions affect code in the VM as well as natively executed code. The very fine-grained permission model and the process isolation provided by Linux ensure the integrity and confidentiality of the system even if the user installs a malicious app. Such malicious software will not be able to invoke API functions if it has not obtained the corresponding permission. Additionally, the application storage is sandboxed as the Linux subsystem checks whether or not the corresponding process has the right to access the data. Due to these access restrictions, accessing other application's data without previously obtaining the permission to do so should not be possible [14].

2.2. Overlays in Android

In this section, we present the necessary basic principles concerning Android's overlays. Overlays are UI elements drawn on top of other applications UI elements, allowing two UI elements to be shown at the same time. We present different types of overlays and elaborate on how the stacking order is determined. Additionally, the terminology of Androids UI rendering system is explained.

Android provides apps with the possibility to create an Activity. Those are acting as the root element for interaction with the user. An `Activity` can be rendered on the screen by creating a `Window` with the UI elements of the app [16]. Each `Window` can have multiple `View` elements [17]. "`View` is the base class for widgets, which are used to create interactive UI components (buttons, text fields, etc.)" [18].

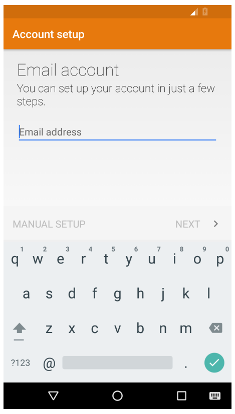
Android provides multiple types of system-wide overlays. Those are `View` elements that are usually rendered on top of windows. The most common one is the `TYPE_SYSTEM_-ALERT` overlay. This type always lays on top of windows and should be used for alerts such as the low power alert. Another one is the `TYPE_TOAST` overlay, which should be used for transient notifications. Those are used to notify a user of messages but only for a short time [19].

Z-Index Each overlay is assigned a z-index by the Android OS. It defines the depth of an UI element in the view hierarchy, meaning the vertical stacking order is determined this way. A higher z-index indicates that the UI element is drawn above an UI element with a lower z-index [20].

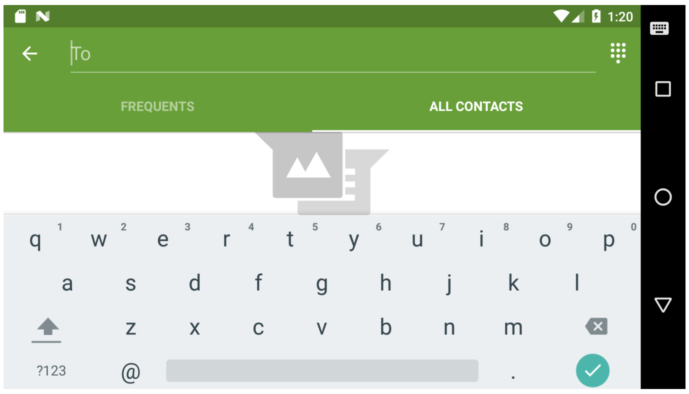
2.3. Keyboard

In this section, we provide background information about keyboard operational modes, such as portrait or landscape. Additionally, we present types of keyboard input fields and explain the difference between functional and non-functional keys.

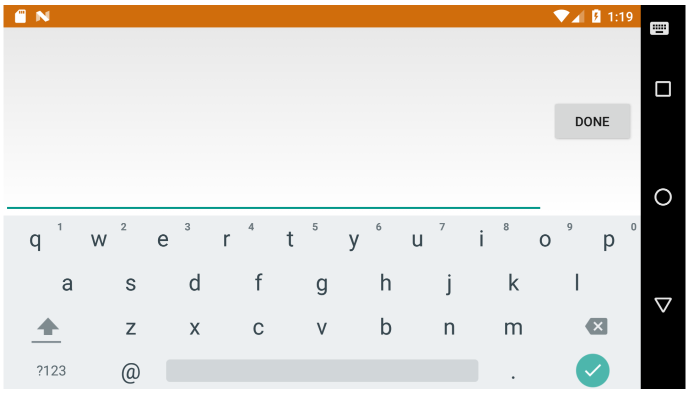
Operational Mode Android provides the user with three modes to enter input. The first mode is the portrait keyboard mode, which is enabled in case the device is in portrait orientation. As seen in Figure 2.1a, the keyboard usually takes up about a third of the screen height and the entire width of the screen. While the device is in landscape orientation, the keyboard adjusts its dimensions to fit the screen in the best way possible. As seen in Figure 2.1b, the keyboard decreases its height compared to the portrait mode. To fit all the keys on the screen, the width is fully used again. In case the input field flag `NoExtractUi` is not set, the keyboard has the permission to enter full-screen mode if it is in landscape orientation [21]. As shown in Figure 2.1c, the `ExtractUi` keyboard mode takes up the entire available space. As the `ExtractUi` keyboard provides an input field called `EditText` in its own UI, this mode is different from the others. This special `EditText` is called `ExtractEditText` and behaves just like a normal `EditText` [22].



(a) AOSP keyboard portrait mode.



(b) AOSP keyboard landscape mode.



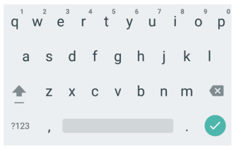
(c) AOSP keyboard landscape `ExtractUi` mode.

Figure 2.1.: Comparison of AOSP keyboard modes.

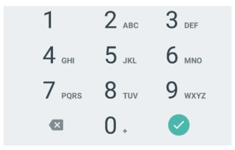
Type Concerning the Android operating system, the `EditText` is the most widely used UI element for input. An `EditText` provides various methods and extends the `TextView` class to inherit most of the numerous attributes that define the design of a `TextView` and additional properties. The `inputType` attribute is used to determine which input type should be entered. For instance, some types are:

- `datetime` – Used for entering date and time
- `numberPassword` – A numeric password field
- `textPassword` – An alphanumeric password field
- `textVisiblePassword` – An alphanumeric password that should be visible
- `phone` – Used for entering phone numbers
- `textEmailAddress` – Alphanumeric input that represent an E-mail address [23]

The amount of keys on the keyboard, the assigned letters and the dimensions of keys are changing according to the input field that is currently focused. This can be seen in Figure 2.2a, showing alphanumeric keys, which is compared to Figure 2.2b showing only numeric keys.



(a) Main page for `textPassword` input fields

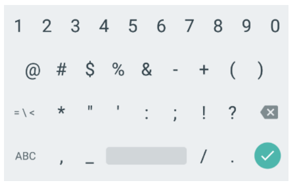


(b) Main page for `numberPassword` input fields

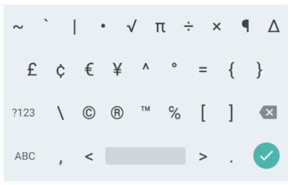
Figure 2.2.: Comparison of main pages for PIN and alphanumeric AOSP keyboard layout.

Function Keys All keys that have functions different to the ability to enter characters are called function keys. This includes, for instance, the Shift key or the Backspace key.

Pages A keyboard can have multiple pages. Those are used to group certain keys together. An additional benefit of grouping is the increased size of the keys as they are distributed on multiple pages. The keyboard pages are shown via the example of the Android Open Source Project (AOSP) keyboard in the following figures. The main pages were already illustrated in the Figures 2.2a and 2.2b. Each page can have a different amount of keys and usually has different letters assigned to the individual keys (shown in Figures 2.3a, 2.3b and 2.4). The dimensions of the keys might therefore change as well. Understanding the behavior of keyboards including their operational modes and ability to adapt to input fields is essential for development of an end-to-end attack. The presented background knowledge will be utilized in Section 5.



(a) First special characters page



(b) Second special characters page

Figure 2.3.: Comparison of AOSP keyboard pages for `textPassword` input fields.

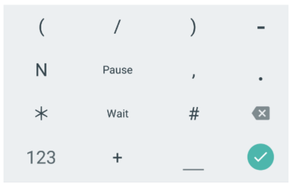


Figure 2.4.: AOSP keyboard Special characters page for `phone` input fields.

3. Related Work

In this chapter, we present related user interface attacks and discuss proposed defense techniques for the “Invisible Grid” attack.

3.1. Attacks

UI attacks exploit weaknesses in the interface in order to deceive the user [10]. “When multiple applications or websites (or OS principals in general) share a graphical display, they are subject to clickjacking (also known as UI redressing) attacks: one principal may trick the user into interacting with (e.g. clicking, touching, or voice controlling) UI elements of another principal, triggering actions not intended by the user” [10]. The OS allows two or more principals to be displayed at the same time to enable Multitasking and to process tasks more efficiently [24]. One example of an attack is tricking the user into clicking a “Buy” button on some online marketplace without the user noticing by overlaying it with an innocuous UI element. These attacks are summarized under the term Clickjacking [10] or more generally UI Deception [11] attacks. There exist many possibilities to exploit UI weaknesses ranging from spawning a fake view in the right moment in order to hijack an input field to drawing overlay views that are actually forwarding clicks to underlying UI elements [25, 10, 26].

In this thesis, we focus on the Keylogger presented in the “Cloak and Dagger” paper [1], which exploits a vulnerability in the Android UI. The attack is able to log all keystrokes that are typed on the virtual keyboard [1].

Zheng et al. [27] show that the “Keystroke Inference #3” from the “Cloak and Dagger” paper is feasible using only `TYPE_TOAST` overlays. This enables an adversary to launch the attack without being dependent on the `SYSTEM_ALERT_WINDOW` permission. However, the authors have not provided solutions to important technical challenges like keyboard detection and layout retrieval, which are addressed in this thesis.

Ulqinaku et al. [28] presented a similar attack in their paper “Using Hover to Compromise the Confidentiality of User Input on Android”. They were able to record keystrokes system-wide by creating overlays on the screen after users clicked. Those overlays then received “post-tap hover events” which were used to infer the clicked key on the keyboard. It was not clarified how the layout retrieval, which is essential for mapping a click to a keyboard key, was done. The authors propose to prevent overlay dimensions of zero and limit transparent overlays to system services only. This approach, however, will most likely interfere with legitimate apps and therefore limit the functionality.

3.2. Defense Methods

There were numerous attempts to develop countermeasures for general UI attacks. For instance, some approaches work by warning the user of malicious activity using an indicator similar to the green Hypertext Transfer Protocol Lock in browsers [12]. Fernandes et al. [11] show that this approach suffers from race conditions and can therefore be circumvented. Other defense methods aim at detecting UI attacks prior to app deployment [13]. Those methods have to be implemented by the app stores and therefore must be highly scalable. The goal is to detect malware before it is added to the official app store. However, detecting malicious use of UI elements is very challenging as there is a variety of legitimate use-cases as well. Furthermore, some defense methods are dependent on OS modification [11] which can only be implemented by the mobile device vendor. Many Android versions are no longer supported by the specific vendor, hence, those defense methods will not be implemented.

In 2012, Hill proposed in his paper "Adaptive User Interface Randomization as an Anti-clickjacking Strategy" [29] to randomize the UI. This approach can be applied to prevent the "Keystroke Inference #3" attack. By randomizing the location of keys on the keyboard mapping of clicks to corresponding keys is no longer feasible. This approach, however, will have a significant impact on usability. The typing speed will be drastically reduced and it's exhausting to search for the desired key. Due to these severe drawbacks, we consider this approach not applicable.

Aljarrah et al. [30] proposed in 2016 a novel defense technique to prevent UI attacks. The defense method called "Window Punching" tries to inject clicks using the `Instrumentation` API from Android. The OS does not allow injection of clicks into UI elements of foreign applications. Hence, if the injected click hits an overlay, an access violation exception is thrown. This circumstance can be used to detect if overlays are obscuring the application. Kalysch et al. [31] proposed in 2018 to use the "Window Punching" defense technique to prevent the "Keystroke Inference #3" attack. This, however, is not possible due to the fact that overlays from the attack are pass-through, hence, no clicks can be received by the overlays. Therefore, the access violation exception will never be thrown resulting in the defense technique being useless. The second method which was also proposed by Kalysch et al. [31] is the use of in-app keyboards with certain flags enabled. They propose to set the `FLAG_SECURE` and disable all accessibility events for the in-app keyboard. We conducted some tests and were able to successfully launch our developed "Keystroke Inference #3" attack despite implementing the proposed defense technique. Hence, "Window Punching" and in-app keyboards, with the `FLAG_SECURE` set, are not sufficient defense techniques.

We conclude, that important technical challenges like keyboard detection and layout retrieval remain unsolved. We, therefore, investigate the feasibility of solving those challenges in this work. The proposed defense mechanisms so far either have a severe negative impact on the usability of the device or are not applicable at all. Due to the lack of working defense mechanisms preserving usability, we present our novel defense technique called `OverlayShifter`.

4. Problem Statement

In this chapter, the basic principle of the “Keystroke Inference #3” attack from the “Cloak and Dagger” paper [1] is introduced.

Android has the ability to draw views on top of other applications. These views are commonly called overlays. One use-case of overlays is, for instance, that the app wants to inform the user of important events but it is currently not in the foreground. If an application wants to draw overlays, it has to access the View Manager and invoke the `addView()` method [32]. This method is protected by the `SYSTEM_ALERT_WINDOW` permission in case an application wants to add a `TYPE_SYSTEM_ALERT` overlay. For apps installed using the official play store, this permission is considered an install-time permission, hence, it’s granted without the consent of the user. “If the app targets API level 23 or higher, the user must explicitly grant this permission to the app through a permission management screen” [33]. Developers, however, can provide an *advance notice* to Google to request an automatic grant of the permission without user interaction [34]. By providing an *advance notice*, apps that are considered trustworthy by Google save their users the effort of manually granting the permission.

Once the malicious app obtained the `SYSTEM_ALERT_WINDOW` permission, it is able to draw overlays on top of every Android app [35]. These overlays can be transparent and therefore invisible to the user. Each view is represented by the `View` class, in the Android system. Android’s View class provides an `onTouch()` event that can be registered. This `onTouch()` event can capture clicks and provide information about them, e.g. x and y coordinates [18]. By declaring the flags `FLAG_NOT_TOUCHABLE` and `FLAG_NOT_-FOCUSABLE`, every click that reaches the View can be passed on to the underlying instance. This results in the following behavior: if an overlay is clicked, the click is passed to the underlying view. Even if the full screen is covered with a transparent overlay, the user is still able to interact with the underlying UI. This implies that the transparent overlay will no longer receive `onTouch()` events by definition of the `FLAG_NOT_TOUCHABLE` flag. The clicks are now passed on to the underlying instance.

By additionally declaring the `FLAG_WATCH_OUTSIDE_TOUCH` the overlay view’s `View.OnTouchListener` is triggered with an `MotionEvent` if the click is outside the area of the view. As the overlay passes each click to the underlying instance, clicks are always outside of the area. The overlay is now able to detect when a click occurred [36]. Even though clicks outside the area are detected, the x and y coordinates of the click can not be queried. Otherwise, this could lead to a massive security breach and the exposure

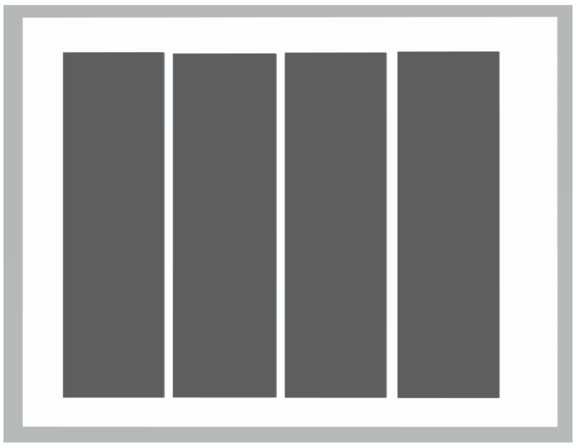


Figure 4.1.: x-y projection of the views

of confidential data. By making a correlation between a key on the keyboard and the click's location, for instance, a malicious app could determine which key was pressed and therefore eavesdrop on sensitive input.

Every `MotionEvent` delivered by a `View.OnTouchListener` has the attribute `OBSCURED`. "This flag is set to true if and only if the click event passed though a different overlay before reaching its final destination" [1]. The name `OBSCURED` is misleading. The function just checks whether or not the click passed through an overlay with a higher z-index without considering the areas. As a result, even if the respective x-y projections of the overlays are disjoint, the top overlay will still obscure the bottom one. The Android operating system even sets the obscured flags for overlays that set the pass-through flags [37].

Therefore, we propose the following definition of `OBSCURED`: Let O_i be an overlay with $i \in \mathbb{N}_0$. Additionally, let $j \in \mathbb{N}_0$ be an integer. Furthermore, let p be the point clicked and denote $p \in O_i$ if p lies in the area of O_i (projected onto the x-y-plane). Additionally, we define z as the injective function that maps an overlay to its corresponding z-index.

$$O_i.\texttt{OBSCURED} = 1 \iff [\text{There exists } O_j \text{ with } z(O_j) > z(O_i) \text{ and } p \in O_j]$$

The `OBSCURED` flag has the value 1 if the click passed through an overlay with higher z-index, otherwise 0. If overlay views are stacked on top of each other, they all inherit different z-indices as shown in Figure 4.2. The z-index determines the stacking order of elements. If each overlay has pairwise disjoint x-y projections, they form a grid in which no overlay overlaps another (Fig. 4.1). If the malicious app now receives `onTouch()` event callbacks, it is able to count the sum of obscured flags for all of its drawn overlays. This can be done by counting all views that have the `OBSCURED` flag set to 1. The sum of `OBSCURED` flags is identical to the index of the clicked overlay, hence an adversary is able to determine which overlay was clicked (Fig. 4.3).

This flag enables numerous attacks. One of them is the logging of keystrokes which draws an overlay on each key of Android's virtual keyboard, also called soft keyboard. Each overlay has a unique z-index. The overlays only cover one key of the keyboard. As a result, the overlays constitute a grid and therefore inherit pairwise disjoint x-y projections. An attacker can now effectively determine which overlay was clicked (fig. 4.3) and eavesdrop on passwords and other sensitive input [1]. In order to make this attack work, however, each overlay needs to obscure no more than one keyboard key. Otherwise, it is not possible to distinguish which key has been clicked. If an overlay is on top of two keys, for instance, an adversary has no way to determine which of the two keys was clicked.

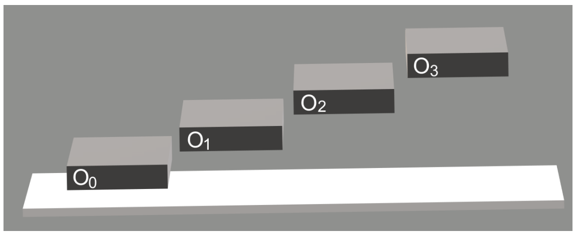


Figure 4.2.: Side perspective showing the z-indices of the Views

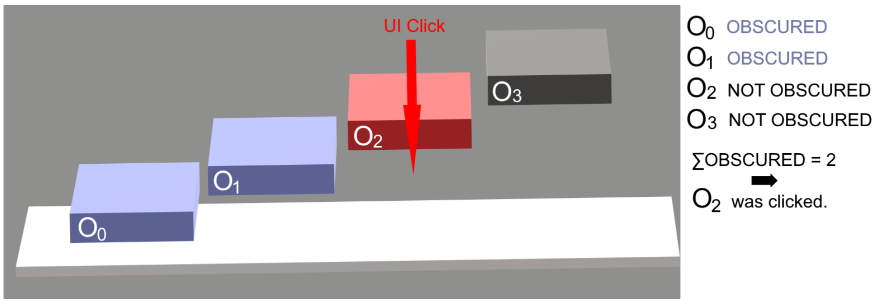


Figure 4.3.: Obscured Views are marked blue in case the red View was clicked

Consider overlays $O_0, ..., O_n$, with disjoint x-y projections, ordered by increasing z-index (fig. 4.2). Again let p be the point clicked. Assume $p \in O_i$. For all $j > i$ we have $O_i.\texttt{OBSCURED} = 0$ because $p \notin O_j$. Due to $z(O_j) < z(O_{j+1})$ for all $j \in \mathbb{N}_0$, we can deduce the definition of `OBSCURED` to:

$$O_j.\texttt{OBSCURED} = 1 \iff p \in O_{j+1} \lor p \in O_{j+2} \lor ... \lor p \in O_n.$$

Which is, due to $p \in O_i$, equivalent to the following:

$$O_j.\texttt{OBSCURED} = 1 \iff j < i$$

Combining, we observe:

$$\sum_{j=0}^{n} O_j.\texttt{OBSCURED} = i$$

Common Vulnerabilities and Exposures Entry The previously described problem was acknowledged as a vulnerability and was recorded in a Common Vulnerabilities and Exposures (CVE) database under *CVE-2017-0860* identifier. It was subsequently patched on almost all Android versions, and to date, only those vulnerable to *CVE-2017-0860* are still at risk [38]. The vulnerability's patch simply sets the `OBSCURED` flag to 0 in case the view which receives the `onTouch()` event set the `FLAG_WATCH_OUTSIDE_TOUCH` [39].

Android security fixes need to be distributed by each vendor individually as they all use customized Android operating systems. Many vendors only deliver updates to their devices for a relatively small period of time. Everyone who is in the unfortunate situation to have purchased smartphones from vendors that no longer support their devices has no access to these security fixes. This leaves their devices defenseless to vulnerabilities that can be exploited at any time [40]. According to StatCounter [3], about 23.5 percent of Android users worldwide still use Android 7.0 or below (see Appendix A.1a for more details). In Asia, it is even more than a quarter of the users (cf. Appendix A.1b). Those devices, running 7.0 or older, potentially have not received security patches, depending on the time they were bought and the manufacturer, and are therefore vulnerable to *CVE-2017-0860*.

For example, the fix for *CVE-2017-0860* for Google Nexus smartphones was distributed by Google in the Security Bulletin November 2017 [41]. The following Android versions were patched: 5.0.2, 5.1.1, 6.0, 6.0.1, 7.0, 7.1.1, 7.1.2. Google Phones only receive security updates for about three years after being introduced. As a result, all phones that are older than November 2014 have a high chance of being vulnerable to *CVE-2017-0860* as they have not received security updates. Nexus 4 was introduced in 2012 and was updated to Android 5.1 as of April 2015 [42]. The security fix support for this device ended in November 2015 [43]. This is just one example of a vulnerable phone. There are many other smartphones that are still used despite the fact that they are no longer entitled to security patches.

5. Approach

In this chapter, we present our approach for implementation of the "Invisible Grid" attack. We define a precise adversarial model for the attack. Furthermore, the technical challenges, for instance, keyboard layout retrieval and keyboard presence detection, for an implementation are analyzed and explained. Based on the model and technical challenges we introduce our implementation structure and elaborate design choices.

5.1. Adversarial Model and Requirements

In the following, we present the adversarial model and certain requirements for a successful attack implementation.

Adversarial Model We assume that the attack app was installed on the victim's Android device and is running in the background. This is a common requirement for attacks on Android phones [28]. Additionally, we require that the device is vulnerable to *CVE-2017-0860*. In order to be inconspicuous, the malicious app should work only using the `SYSTEM_ALERT_WINDOW` permission. We assume that this permission is already granted by the OS or the user. We furthermore assume that the user only enabled one keyboard at a time. Hence, the attack implementation does not need to support multiple keyboards simultaneously. In particular, we assume that the user does not switch between different keyboards during the runtime of the attack. Furthermore, it is assumed that the keyboard always has a width identical to the screen's width.

Requirements In the following, we present requirements for a successful attack implementation. The app should launch the attack in the background without the user noticing. As a result, we require that the app should act as inconspicuously as possible. The attack must therefore be completely invisible to the average user. The attack itself must be reliable to be able to eavesdrop on sensitive input. Almost all keyboards have multiple pages as it is hard to fit all keys on one single page. These keyboards usually have a function key, which is used to switch between independent pages, as seen in the lower-left corner of Figure 2.2a. As a result, we require that an adversary needs to support keyboards with multiple pages. Additionally, many keyboards provide upper- and lowercase letters. Shift keys, therefore, have to be supported as well. This entails, that an implementation must monitor the state of the shift key and adapt the attack layout if it is switched to another page.

5.2. Technical Challenges

In this section, we analyze the technical challenges that need to be solved to make the "Invisible Grid" attack by Fratantonio et al. in the "Cloak and Dagger" paper feasible [1, 2]. We label challenges with the letter **C** combined with their consecutive number. Additionally, we present arguments which demonstrate the necessity of certain challenges.

Keyboard Layouts [C1] To successfully launch the "Cloak and Dagger" attack, the correct key labels, and, more importantly, their positions must be obtained. The layout is necessary for the attack, as simply drawing any grid has two major disadvantages. As a first disadvantage, the grid must be very narrow as touches can otherwise not be uniquely assigned to a specific key. The input text might therefore not be unambiguously reconstructed. Hence, an adversary has to guarantee that a grid field can be attributed to one single key. If this condition is fulfilled, the amount of possible texts is reduced to one. In order to achieve this, a very narrow grid is required to guarantee that one grid field does not occupy two keyboard keys. However, the drawing of a very narrow grid field requires a high number of overlays and therefore consumes a large number of resources. The increasing utilization of resources results in higher loading times that will be noticeable by the user.

We conducted a performance test of the Android Choreographer, using the `systrace` tool, which coordinates the rendering of the contents of an app [44]. As seen in Table 5.1, the **Total** time required to render the applications increases with each overlay. We measured the time required to load an E-mail app with different amounts of overlays on the screen. As seen in the Table the **Total** necessary to load the app is 374.57 ms without any overlays. However, by adding 200 overlays the time heavily increases to 1,127.98 ms. We only present expressive steps from the measured performance of the Choreographer to preserve clarity. The column **doFrames** specifies the time necessary to render the frame for the application. **Traversal** specifies the time necessary for the Android OS to traverse the layout hierarchy. This must be done to ensure that parent UI elements are drawn before child UI elements. **Draw** specifies the time which is necessary for the actual drawing process of the rendered frame. The process of loading the layout from a eXtensible Markup Language (XML) definition is grouped under **Inflate**. We can clearly see that the drawing and inflation process is only slightly impacted by the amount of overlays (7.16 ms and 18.52 ms for drawing / 35.07 ms and 35.14 ms for inflation). Additionally, we can see that the **doFrames** and **Traversal** steps are heavily increasing with 155.94/153.10 ms versus 449.04/450.95 ms when 200 overlays are present.

We have shown that the time necessary to render any app heavily increases with the amount of overlays. This, therefore, makes the attack not feasible for narrow grids due to the user noticing severe slow-downs. Most European keyboards based on Latin script use at least 26 keys from the alphabet [45]. This sets a lower limit on the number of overlays that have to be drawn as every key needs an independent overlay to assemble the attack and functional as well as special character keys further increase that number, for instance, the English AOSP keyboard as seen in Figure 2.2a has 33 keys. As shown in the Table adding 35 overlays only slightly impacts (92.59 ms) the loading time of applications. Hence, by only utilizing the lowest required amount of overlays the attack is feasible.

The other disadvantage is that by drawing just any grid, it is not possible for an adversary to identify which character the key represents. An adversary has to attribute each grid field to a character on the keyboard after the actual attack took place. This can be very challenging as additional information about the device, such as keyboard language, screen resolution and device orientation, are necessary in order to infer the label of keys. Additionally, the layout and labels can change during recorded input. An adversary, however, needs to know these attributes at every moment of the capturing process. Due

Amount of Overlays	doFrames (ms)	Traversal (ms)	Draw (ms)	Inflate (ms)	Total (ms)
0	155.94	153.10	7.16	35.07	**374.57**
35	210.60	206.86	11.67	35.11	**467.16**
100	260.36	258.50	15.16	34.97	**566.32**
200	449.04	450.95	18.52	35.14	**1,127.98**

Table 5.1.: Selected steps from Choreographer performance for different amount of overlays

to the complexity behind the keyboard layout, heuristics seem to be out of reach and cannot effectively recover the input text.

As Android does not provide an API for retrieval of the keyboard layout, a custom approach needs to be developed. The approach has to work dynamically and support any keyboard. Furthermore, the attack has to be independent of the device orientation and therefore has to work in portrait as well as landscape mode.

Keyboard Presence [C2] and Keyboard Orientation [C3] To perform the attack, an adversary has to know in which orientation and layout the overlays have to be drawn. Another challenge is therefore the detection of keyboard presence. An approach has to work system-wide as sensitive input is usually not entered in the app of the adversary. This implies that the detection must be feasible without restricting to a specific app or configuration. As already mentioned in **C1**, both device orientations should be supported. Furthermore, the three different keyboard modes (as shown in Section 2.3) should be supported as well as sensitive input can be entered using any of the modes. Some keyboards change their internal state when closed and reopened, for instance, the AOSP keyboard resets the shift key to disabled for each subsequent reopening. A continuous stream of information about the keyboard's presence, including the device orientation, is therefore necessary. Thus, an approach has to detect the appearance or disappearance of the keyboard as fast as possible. For clarity, detection of the current device orientation is grouped as the abbreviation **C3**.

Keyboard Type Detection [C4] Another challenge is the detection of the current keyboard input type. Android provides numerous types of input fields. Some of them only support numeric inputs while others are not restricted at all, e.g. by accepting alphanumeric keys or special characters. Many keyboards have a feature that adjusts the shown keys to the corresponding input field [46]. Among different input types, the character assignments to the keys may change or additional keys may be added. Thus, it is required to infer which characters were typed through knowledge of the input type.

Window Manager Performance [C5] Android's Window Manager is responsible for adding overlays to the screen. A measurement has shown that it takes up to 67 milliseconds to draw 20 overlays as presented in Figure 5.1. The measurement was performed by drawing a number of random overlays and repeating this process ten times. The time was measured by saving the starting timestamp and comparing it with the timestamp after the Window Manager call finished. A distribution summary, i.e. boxplot of those ten iterations, was then calculated to have a representative result. As placing 30 overlays takes up to 94 milliseconds, it takes too much time to reliably eavesdrop on sensitive input. As soon as the keyboard appears, most users almost instantly start typing. If the overlays are only added after a hundred milliseconds, an adversary may miss keystrokes. Therefore, the overlays should be kept on the screen to avoid the time-consuming drawing process for each successive reopening. This, however, entails other implementation specifics that are

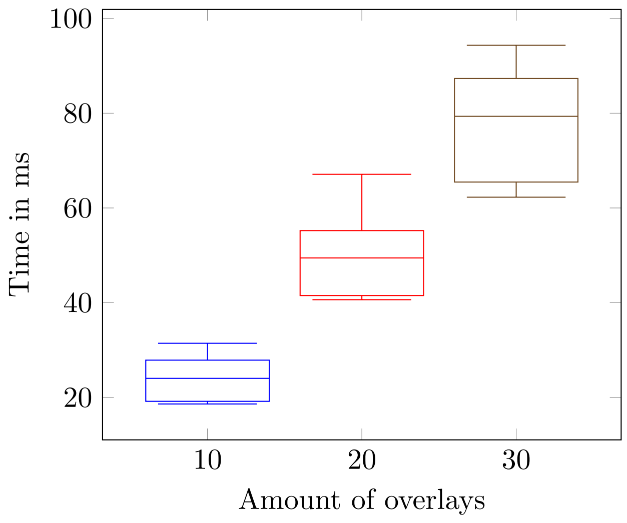


Figure 5.1.: Time needed in ms to place overlays

further elaborated in Section 5.3.2.

5.3. Attack Design

In the following section, we propose abstract solution ideas to the challenges formulated in Section 5.2. We begin with presenting a control flow graph of the attack, then discuss individual attack steps and propose a class structure for an attack implementation.

5.3.1. Attack Control-Flow Graph

In order to fulfill that the attack is invisible to the average user, the malicious app is launched in a minimized window state and utilizes transparent overlays.

The control-flow of the app is structured as shown in Figure 5.2. The first step is to determine the dimensions, i.e. width and height, of the keyboard. This information is necessary for the creation of an attack layout as required by **C1**. A module called *KeyboardSizeProvider* is therefore responsible for returning the keyboard's dimensions. A keyboard can have multiple measurements depending on the current device's orientation and the mode it is operating in. As soon as this information is gathered, the execution continues by providing the values to a module named *KeyboardLayoutProvider*. In our approach, this module is implemented via two different methods: the layout is either read by the *ResourceProvider* from a previously defined layout file that was created by an attacker prior to execution or it is dynamically extracted from a keyboard by the *DynamicProvider*. Different methods are hereby used to obtain layout files that belong to the keyboard in order to obtain information about the layout. The static approach (*ResourceProvider*) acts as a fall-back solution in case the dynamic extraction (*DynamicProvider*) of the layout fails. The dynamic extraction fails in case the user utilizes keyboards different from the supported ones.

Simultaneously, a module named *KeyboardPresenceProvider* is instantiated to detect the keyboard state, which contains information about whether or not the keyboard is actively shown on the screen or hidden (**C2**) as well as the keyboard's orientation (**C3**). This information is vital as an adversary has to know at what point in time and in which orientation, i.e. landscape or portrait mode, the overlays have to be drawn. Furthermore, the used input type is provided to fulfill challenge **C4** as well.

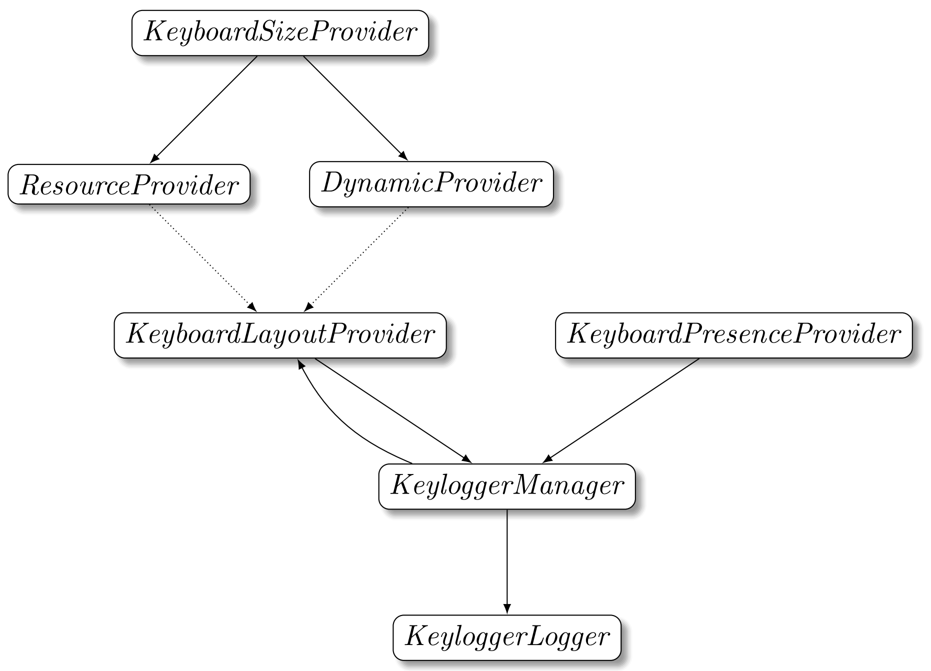


Figure 5.2.: Control-flow graph of the attack. Edges represent successive steps of the control flow. Dotted edges represent the relation to a superclass.

The data from the *KeyboardLayoutProvider* and *KeyboardPresenceProvider* are used to proceed with the execution of a module called *KeyloggerManager*, which represents the attack's core. It is responsible for merging all previously gathered data in order to perfectly deploy the attack on a device. This module fetches the keystrokes and forwards them to the next module called *KeyloggerLogger*. In particular, state transitions initiated by the shift key are handled by the latter module. On the other hand, keys responsible for switching to different layout pages are handled by the *KeyloggerManager* as it is in charge of drawing the attack layout. The *KeyloggerManager* is also responsible for adapting the attack to changes in the environment, for instance, device rotations from portrait to landscape mode during the attack.

5.3.2. Communication Between Modules

As the app does not have information about the keyboard state in advance, an asynchronous stream of information from *KeyboardPresenceProvider* to *KeyloggerManager* is desired. The asynchronous approach is superior to the synchronous one as, for instance, changes in the orientation of the device cannot be foreseen. If a synchronous approach is used, a polled operation is necessary [47]. This produces large overhead of resources as polling of the current keyboard mode is executed in regular intervals. Redundant information is produced, for instance, when nothing changed but a polling operation occurs.

KeyboardLayoutProvider and KeyloggerManager By using asynchronous methods, the information is provided only once, using a callback in case it has changed. This implies that either:

a) *KeyloggerManager* needs to fetch relevant layouts from the *KeyboardLayoutProvider* or

b) the *KeyboardLayoutProvider* sends all possible keyboard pages.

Assuming b), the *KeyboardLayoutProvider* sends all possible keyboard pages in the initialization step of the attack. This leads to longer loading times as all keyboard pages must be provided initially. Furthermore, some keyboard pages may not be necessary for the attack and will therefore produce an overhead. As approach a) is implemented, the *KeyboardLayoutProvider* has to provide a method for adapting the layout to new keyboard modes.

This means the communication of those two modules is bidirectional. For instance, in case a change of orientation from portrait to landscape mode occurs, the keyboard layout must also change, to the corresponding orientation. Otherwise, the attack layout is still in portrait mode while the user types his sensitive information in landscape mode. This produces false results. In case a user changes the keyboard page, e.g. to the page with special characters, the overlays attack layout must also adapt correspondingly. Otherwise, it is not synchronous with the underlying keyboard layout, producing false results as well.

KeyboardPresenceProvider and KeyloggerManager Keyboard layouts usually differ for each input type. For instance, the keyboard used to enter Personal Identification Number (PIN)s only contains numeric keys while the user also enters letters in case of an alphanumeric password. Once again, the attack overlays must be identical to the keyboard layout. As a result, it is essential that the input type must be detected (**C4**). This information is gathered by the *KeyboardPresenceProvider*. The current keyboard type must be provided every time the *KeyloggerManager* queries the *KeyboardLayoutProvider* for a layout. The *KeyboardPresenceProvider* must therefore inform the *KeyloggerManager* of any changes that occurred.

As described in Section 5.2, re-drawing the overlays every single time consumes too much time (**C5**), this problem has to be addressed as well. One solution is to keep the overlays on the screen regardless of the state the keyboard is in. This leads to the problem that clicks are detected regardless of the keyboard state. If the keyboard is closed, for instance, and the user clicks on an overlay it is considered a click on a keyboard key. As a result, a feature is necessary that is able to verify if clicks occurred on the keyboard or in some other app. The verification feature does not require another module since the *KeyboardPresenceProvider* delivers exactly this information. In case a click occurs and the keyboard is opened, the malicious app can infer that the click was inside the area of the keyboard. Hence, by solving the technical challenge **C2** the challenge **C5** is fulfilled too.

In this chapter, we have defined an adversarial model and discussed requirements for an attack implementation. Furthermore, we have elaborated on the technical challenges and proposed solution ideas for our end-to-end attack implementation. The feasibility of our solution ideas is investigated in the next section. Furthermore, we present specific implementation approaches for individual classes of the control-flow graph.

6. Implementation

In this chapter, we present in-depth information about the implementation of the concept defined in Section 5.3. Additionally, we justify our choices of implementation by elaborating the strengths and limitations of different approaches and analyzing the feasibility of each module.

6.1. Android Prerequisites

The following paragraphs explain the basic principles to comprehend the developed approaches.

Input Method Android provides an interface `InputMethod` to enable communication between the keyboard and apps that are dependent on user input. An `InputMethod` is not restricted to keyboards but can, for instance, perform speech to text conversion as well [49]. However, due to the nature of the attack, the focus is on keyboards used as an `InputMethod`. This interface is a highly security-critical aspect of the OS and therefore has a special architecture with the goal of preserving confidentiality and integrity. One of those architectural properties is that only the Android OS is able to access the `InputMethod` interface by demanding a signature permission in order to access it.

The interface itself can be used to create new sessions between apps and the keyboard. If an adversary controls the session between an `InputMethod` and an app, he is able to log all input events.

Another security feature to guarantee integrity and confidentiality of the input is that the `InputMethod` is only able to communicate with one app at a time. As a result, only the currently active app is able to access an `InputMethod` [50].

Input Method Info The malicious app is able to obtain a new instance of the `InputMethodManager` by executing the `getSystemService()` method. The user can choose to enable or disable certain `InputMethods` from a list of installed input methods. All enabled input methods can be obtained by calling the `getEnabledInputMethodList()` function [50]. Android stores the unique IDentifier (ID) for currently selected `InputMethod's` in the secure settings. The settings are considered secure as they are read-only and additional permissions are required in order to modify them [51]. However, they can be read without permissions.

As a result, the ID `DEFAULT_INPUT_METHOD` can be read without any further permission. The malicious app can get information about the currently enabled `InputMethod` by iterating over all input methods and compare the ID. The fact that an `InputMethod` is enabled does not reveal anything about its state, e.g. “open” or “closed”.

Input Method Subtype Each Input Method has a certain number of `InputMethodSubtypes`. The “Subtype can describe locale (e.g. `en_US`, `fr_FR`, ...) and mode (e.g. voice, keyboard, ...)” [52]. The class is therefore used to specify which language is currently in use. The malicious app can obtain the current Subtype by executing the `getCurrentInputMethodSubtype()` method from `InputMethodManager` [53].

Window Manager The first step of adding overlays is to obtain the current instance of Android’s `WindowManager` by executing the `getSystemService(Context)` function with `Context.WINDOW_SERVICE` as an argument [36].

Window Manager Layout Parameters In order to add an overlay, some layout parameters have to be defined first by declaring new `WindowManager.LayoutParams`. By changing the `x` and `y` properties, the location of an overlay can be set. Additionally, flags, such as `FLAG_NOT_TOUCHABLE`, dimensions using width and height properties and the overlay type, such as `TYPE_SYSTEM_ALERT`, can be declared. In the next step any `View` and the previously declared `WindowManager.LayoutParams` can be passed to the `addView(View, LayoutParams)` method of the `WindowManager` in order to draw a new overlay on the screen [36]. The `TYPE_SYSTEM_ALERT` flag is deprecated as of API level 26 (Android 8.0) [54]. Due to the focus on Android 7.0, the flag is still usable for our implementation.

As described in Chapter 4, an attack requires the flags: `FLAG_NOT_TOUCHABLE`, `FLAG_NOT_FOCUSABLE` and `FLAG_WATCH_OUTSIDE_TOUCH` to be set. Additionally, a pixel format that supports transparency can be declared by setting the property `PixelFormat.TRANSPARENT`. Overlays with pass-through behavior are always considered outside of touch. By using these flags, an adversary can create transparent pass-through overlays that are always notified in case a click occurs.

6.2. Package Diagram

In the following, we elaborate on the Java package diagram based on the control-flow graph. Appendix Figure B.2 shows the package diagram including classes and interfaces. Most of the classes are structured as specified by the control-flow graph. The *KeyboardLayoutProvider* is defined by an interface to acquire uniform methods and to provide safe access without typecasting. As a result, each *LayoutProvider*, e.g. the *DynamicProvider* and *ResourceProvider*, implement certain methods that ensure support for desired operations, e.g. the retrieval of a layout in case the keyboard orientation changes. Additionally, the use of interfaces makes the *KeyboardLayoutProvider* extensible if a new method to retrieve the layout emerges.

All asynchronous operations are implemented similar to the Observer pattern by using interfaces [48]. If the *KeyboardPresenceProvider* detects a change of orientation, the callback *OrientationCallback* is notified. The same procedure happens if a height change is detected. But this time, however, the *ResultCallback* is notified. Due to the reasons elaborated further on, we chose to additionally provide a callback if the results from the *KeyboardSizeProvider* are available. By using interfaces we can take advantage of the easy access to information. If a module needs information from one of those providers, we can effortlessly implement the interface and will be notified automatically in case a change occurs.

6.3. Keyboard Presence

As Android does not provide an API call to query information about the presence of a keyboard, we present our developed approaches for the *KeyboardPresenceProvider* implementation to achieve this goal. Furthermore, we show the limitations and strengths of the developed approaches. The presence detection is used to gather three essential information pieces about the keyboard:

a) Keyboard State – Determines if the keyboard is opened or closed,

b) Keyboard Type – Determines which type of `EditText` is currently focused,

c) Keyboard Orientation – Determines in which orientation the keyboard is displayed.

6.3.1. State

The basic idea for detecting the current state of the keyboard is to gather the height of the keyboard. This value can then be used to infer the state as a non-zero value indicates an opened keyboard. In particular, we solve the challenge **C2**. The method should work for all three keyboard modes:

1.) Portrait Mode – The keyboard is in portrait orientation,

2.) Landscape Mode – The keyboard is in landscape orientation,

3.) Landscape `ExtractUi` mode – The keyboard is in landscape orientation and has extracted the input field.

We have developed three individual methods to measure the keyboard height. The first method *Reflection Method* works by invoking hidden API functions. The second method *Window Method* aims at creating special overlays that are capable of detecting the keyboard state due to layout changes. The last method called *Advanced Window Method* is a more sophisticated implementation of the second method.

Reflection Method The first proposed by us method takes advantage of the reflection that is supported by Android. "[R]eflection allows inspection of [...] methods at runtime without knowing the names [...] at compile time. It also allows [...] invocation of methods" [55]. Android provides a public API but also maintains a hidden API that is intended for use by the operating system only. The development team behind Android accidentally exposed an API method `getInputMethodWindowVisibleHeight()` that was actually supposed to be annotated as hidden [56]. This method belongs to the `InputMethodManager` and can be used to query the visible height of the keyboard by obtaining the visible rectangle of a keyboard and returning the height of it. Hidden API functions can be accessed using reflection techniques but may be removed without any warning in future Android versions. By obtaining a new instance of the `InputMethodManager` and acquiring the hidden API function by using the `getDeclaredMethod()` function, an adversary is able to make it accessible [57]. After the reflection process, the current height of the keyboard can be obtained by executing the function.

This method works system-wide, meaning it does not matter if the keyboard is opened in the malicious app or any other app. Using this technique the keyboard height in the portrait mode and landscape mode can be obtained. However, we observed that the method does not work for the `ExtractUi` keyboard mode. Another limitation is, that the method needs to be executed in a loop to have continuous information about height changes, which produces massive performance overhead. Due to these limitations, other methods were developed which try to eliminate those drawbacks.

Window Method Due to the limitations of the "Reflection Method", especially concerning the resource overhead, another technique was developed. This method takes advantage

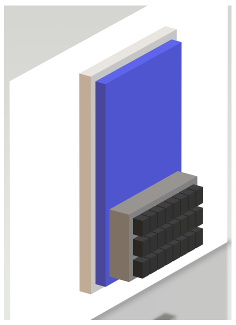


Figure 6.1.: The white rectangle represents an activity. The blue rectangle represents the specially-crafted overlay. It is placed between the keyboard and the foreground activity.

of the fact that the appearance of the keyboard triggers a layout change event. This discovery was first presented in a basic form by Siebe Brouwer in his GitHub repository [58]. However, his method does not work system-wide and was therefore only used as inspiration.

During the implementation, we observed an interesting behavior. By definition, an `TYPE_SYSTEM_ALERT` overlay always resides on top of each activity in Android. However, by adding a special `PopupWindow` to the overlay, the z-indices of all overlays that are currently on the screen are adjusted. The `PopupWindow` has the flag `INPUT_METHOD_NEEDED` set, which specifies that the `PopupWindow` needs to be able to receive input [59]. As a result, all overlays that have lesser z-indices than the currently added overlay with `PopupWindow` are moved on top of every activity but behind the keyboard. To summarize, by adding the `PopupWindow` to an overlay, we can shift the z-indices of the overlays that are currently on the screen to be between activities and a keyboard. This can be seen in Figure 6.1. This behavior will be leveraged as explained in the following paragraphs and we refer to this technique (adding a `PopupWindow` with the mentioned flags to an overlay) as a specially-crafted overlay. Assuming a specially-crafted overlay was already added, we want our adversarial overlay to automatically resize to the same dimensions as the keyboard. Therefore, the flags `SOFT_INPUT_ADJUST_RESIZE` and `SOFT_INPUT_STATE_ALWAYS_VISIBLE` for the `PopupWindow` are also declared and the overlay's type is changed to `TYPE_TOAST` with the flag `FLAG_NOT_FOCUSABLE` set. As a consequence of the flags, the appearance of the overlay now no longer closes the keyboard if it was already showing and resizes its dimensions to fit the keyboard.

Resizing of the adversarial overlay due to state transitions of a keyboard can now be detected by hooking into the `onGlobalLayout()` method. This will cause a listener (`OnGlobalLayoutListener`) to be called if the layout undergoes size changes [60]. If the keyboard is shown, it will trigger a resize of the adversarial overlay and therefore a layout change. If the keyboard is closed, there no longer exists a valid reason to resize and therefore a new layout change is required. This behavior is shown in Figure 6.2. As the goal is to infer the keyboard's state by obtaining the keyboard's height, the overlay's height is set to the screen height. However, the status bar can only be covered by declaring additional flags and is therefore not obscured. This circumstance must be considered and results in the following calculation:

$$\text{KeyboardHeight} = \text{ScreenHeight} - \text{StatusBarHeight} - \text{OverlayHeight}.$$

By setting the width property of the adversarial overlay to zero the method is completely invisible to the user and still triggers the layout change event. The method works system-wide and produces very little resource overhead as no loops are required. The continuous

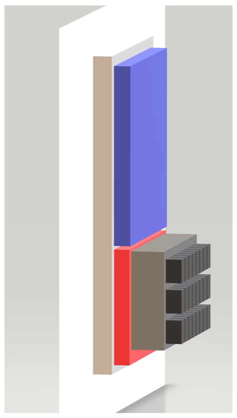


Figure 6.2.: The white rectangle represents an activity. The blue rectangle represents the specially-crafted overlay. The red rectangle represents the area that is occupied by the keyboard.

stream of information about the height is still guaranteed as every state transition of the keyboard triggers a change in layout. In spite of that, one limitation is that this technique is not applicable to the `ExtractUi` mode of the keyboard. As the keyboard occupies the whole screen height and therefore does not resize any overlays as seen in Figure 6.3.

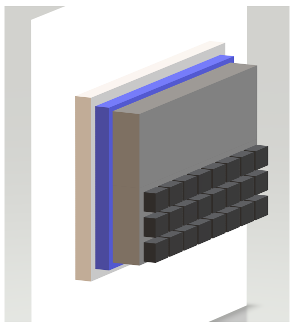
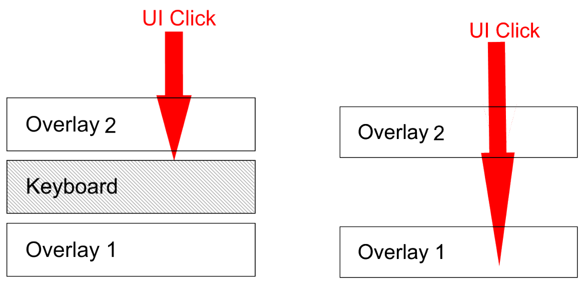


Figure 6.3.: Keyboard in `ExtractUi` mode. The gray rectangle represents the keyboard. The blue rectangle represents the specially-crafted overlay.

Advanced Window Method The third technique was developed by us to combat the limitation of the "Window Method", which does not detect the keyboard state in the `ExtractUi` mode. We assume a specially-crafted overlay O_1 is already added to the screen. The overlay should inherit pass-though behavior and therefore has the flags `FLAG_WATCH_OUTSIDE_TOUCH`, `FLAG_NOT_TOUCHABLE` and `FLAG_NOT_FOCUSABLE` set. We want the overlay to be invisible to the user and therefore the width and height is set to zero.

The second overlay O_2 is a `TYPE_SYSTEM_ALERT` overlay, which will be added after the first one. It has the pass-through flags (`FLAG_WATCH_OUTSIDE_TOUCH`, `FLAG_NOT_TOUCHABLE`,`FLAG_NOT_FOCUSABLE`) set and to be invisible to the user its width and height is set to zero as well.

The approach works as follows: An adversary adds two overlays O_1 and O_2, in this order, to the screen. The order is of great importance as by adding a specially-crafted overlay, all overlays already on the screen will be shifted to the back in terms of their z-indices. Additionally, the call order of `ACTION_OUTSIDE` events is descending, which means the `View` with the highest z-index is notified first if an event occurs. Overlay O_1 is the previously described specially-crafted overlay. The second overlay O_2 is a standard `TYPE_SYSTEM_ALERT` overlay. As the keyboard does not demonstrate pass-through behavior, it



(a) Overlay 1 (O_1) does not receive click events in case a keyboard is opened.

(b) Overlay 1 (O_1) receives click events in case the keyboard is closed or the click did not occur inside the keyboards region.

Figure 6.4.: Schematic sketch of the Advanced Window Method. Overlay 2 (O_2) always receives clicks events.

will not pass `MotionEvents` to the underlying overlay O_1 but consumes them. However, as overlay O_2 is on top of the keyboard, it will indeed receive an `ACTION_OUTSIDE` event. This circumstance can be leveraged to detect the state of the keyboard as shown in Figure 6.4. If overlay O_2 receives an `ACTION_OUTSIDE` event but O_1 does not receive an `ACTION_OUTSIDE` event afterwards, an adversary can infer that the keyboard must have consumed the event. Therefore, its state must be opened as well. On the contrary, if an event is received by O_1, it implies that the keyboard state must be closed or the user did not click inside the region of the keyboard. This behavior can therefore be leveraged to infer the state of the keyboard and if clicks occurred inside the region of the keyboard.

This technique supports all three keyboard modes and works system-wide. As the information can only be inferred in case a click occurred, the requirement for providing continuous height information is violated (**C2**). This drawback is addressed by utilizing a hybrid technique, which is a combination of the “Window Method” and the “Advanced Window Method”. The continuous height information from the “Window Method” can be used to detect keyboard presence in portrait and landscape mode. And the “Advanced Window Method” is used to detect keyboard presence in `ExtractUi` mode.

Keyboard Height Threshold We additionally define a threshold at which the state of the keyboard is considered to be opened or closed. This is necessary due to the navigation bar which also occupies height on the screen and can trigger layout change events. Screens have different densities, hence keyboards can therefore have different absolute pixel heights on devices. The *Android Style Guides* suggest that touch targets should at least be 48 by 48 Density-independent Pixels (dp) [61]. Therefore, a conservative assumption is that the touch targets have at least a height of 30 dp. A standard keyboard usually has at least 4 rows. We can now calculate the threshold at which we consider a keyboard to be opened:

$$\text{Threshold} = 30 \text{ dp} \times 4 = 120 \text{ dp}.$$

Due to the dp units, it is independent of the device and must be converted to pixels on each device by multiplying with the screen density, which can be obtained from the OS.

6.3.2. Type

In this section, inference techniques of the current type of the keyboard are described. In particular, we solve the challenge **C4**. The types can, for instance, be “PASSWORD”, used for entering alphanumeric passwords, or “PIN”, used to enter numeric passwords. The type of the keyboard is determined by the type of the currently focused `EditText`. Android only provides information about the `EditText` in an `EditorInfo` object that is exclusively passed to the keyboard app. As it runs in a different package, there is no legitimate way an adversary can access it without exploiting vulnerabilities [21]. An adversary is able to infer the displayed keyboard layout by querying the `inputType` of the `EditText`. However, this only works in case an adversary can directly access the focused `EditText`. This is only the case if an `EditText` is owned by the malicious app. This implies that there is no official way of detecting which `inputType` is active for the `EditText` that does not belong to the malicious app [18].

Furthermore, it does not appear that the `EditText` can be inferred from the `InputMethod`. The function `dispatchKeyEventFromInputMethod()` is used to dispatch a passed key from the active keyboard to a currently focused `EditText` [53]. Even emitting illegal letters by using this function to the `EditText` cause no detectable exception.

One approach for detecting different layout types of keyboards is to infer it by analyzing the height of a keyboard. The height can be obtained by either using the *Reflection Method* or *Window Method* described in Section 6.3. The two most important layouts for an adversary are the `TYPE_NUMBER_VARIATION_PASSWORD` for PIN inputs and the `TYPE_TEXT_VARIATION_PASSWORD` for alphanumeric passwords. Furthermore, most keyboards have different layouts for the alphanumeric and PIN passwords.

Our approach works as follows: Let $h(x)$ refer to the height of a keyboard x. Additionally, let K_T be a keyboard of `InputType` T. In case the inequality

$$h(K_{T_i}) \neq h(K_{T_j})$$

is true for all $i \neq j$, an attacker is able to successfully determine the `InputType` by height. That implies that the condition

$$h(K_{\texttt{TYPE\_NUMBER\_VARIATION\_PASSWORD}}) \neq h(K_{\texttt{TYPE\_TEXT\_VARIATION\_PASSWORD}})$$

is fulfilled and therefore an adversary is able to successfully determine if the user types a PIN or an alphanumeric password provided that he knows the height of the keyboard that is currently active. The approach can be generalized by mapping heights that were previously captured for certain types to the respective keyboard type. Therefore, the goal is to create an injective lookup table that assigns each keyboard height the respective keyboard type.

The AOSP keyboard does not satisfy the previously mentioned generalization and inequality. The implemented malicious app is therefore not able to distinguish between PIN and alphanumeric passwords in case of the AOSP keyboard. As a majority of passwords are alphanumeric and therefore constitute the greater attack vector, we chose to focus on them in case the inequality is not fulfilled by drawing a grid for alphanumeric passwords by default.

6.3.3. Orientation

Current device orientation can be queried by registering a broadcast from the Android OS called `ACTION_CONFIGURATION_CHANGED`. This process of registering requires no additional permissions as it is done by calling the unprotected method `registerReceiver()` [62]. The broadcast is always sent if the current device configuration has changed. This includes the device orientation [63]. Hence, by implementing this approach we have solved the technical challenge **C3**.

6.4. Keyboard Dimensions

The positions of the keys on the keyboard and the assigned characters will be referred to as attack layout. In order to create keyboard attack layouts, it is necessary to know the dimensions of a keyboard in advance. The purpose of the *KeyboardSizeProvider* module is to provide this information. The dimensions of a keyboard are defined by its width and height. Most Android keyboards expand their width to the screen width, for instance, the AOSP keyboard as seen in Figure 2.2a or Google's Gboard (fig. 7.1d). We, therefore, decided to assume in the adversarial model that the keyboard's width is identical to the screen's width. This assumption should prove right in most of the cases as it makes little sense to vertically restrict the keyboard. Therefore, it is sufficient for this module to provide information about the current keyboard height. Many keyboards have different heights depending on the orientation mode they are operating in. For instance, the height of the AOSP keyboard in landscape mode is much smaller compared to the portrait mode. It is therefore required that not only the keyboard height is provided in advance but also the different orientations of the devices are respected (**Requirement Orientations**). The information provided by this module should therefore be equal to the keyboard height in portrait and landscape orientation.

In order to utilize the keyboard type detection method proposed in Section 6.3.2, the keyboard height for different input types needs to be provided. As a result, one requirement is that the keyboard height for the `TYPE_NUMBER_VARIATION_PASSWORD` and the `TYPE_TEXT_VARIATION_PASSWORD` keyboard is provided (**Requirement InputTypes**).

The main idea is that the malicious app deliberately opens the keyboard to take measurements. Those measurements can then be combined with the knowledge of an input type and therefore a lookup-table can be created. The first step is to display an `Activity` that contains the mentioned `EditTexts`, assuming we added an `EditText` *number* of type `TYPE_NUMBER_VARIATION_PASSWORD` and an `EditText` *alpha* of type `TYPE_-TEXT_VARIATION_PASSWORD`. In order to fulfill requirement **Orientations** the *Keyboard-SizeProvider* module changes the orientation to portrait mode. This can be achieved by calling the Android method `setRequestedOrientation()` [64]. In the next step, the module requests focus for the *alpha* `EditText`. As a consequence of this operation, the currently active keyboard opens as user input is expected for the focused `EditText`. As soon as the keyboard is shown for the focused *alpha* input, the reflection method described in Section 6.3.1 to infer the height is used. It was chosen as it is very fast to utilize and the height value is only needed once. Therefore, the performance overhead is well-acceptable as a one-time cost. Due to the fact that its not known when a keyboard appears, the algorithm queries the keyboard's size until it is greater than the threshold. After a value was obtained, the algorithm then proceeds to perform the same technique for the *number* `EditText` in order to infer the height as specified by requirement **InputTypes**. To comply with requirement **Orientations**, the process is repeated for landscape orientation as well.

The described technique is not invisible on the victim's device as the keyboard opens and closes multiple times. Suspicion may be aroused by tech-savvy users that consider this behavior odd. By utilizing this technique, the requirement stating that the malicious app has to be as inconspicuous as possible is not fulfilled. Most smartphone users, however, won't notice this process at all as it only takes a few seconds and may have legitimate reasons. The automatic opening of the keyboard itself is not a malicious act. Additionally, the taking of measurements could be done when the user does not pay attention to the device, e.g., at night-time or when the device is not moving. We, therefore, consider the approach compliant with the requirements and the defined adversarial model.

6.5. Keyboard Layout

In order to make the attack feasible, an adversary needs the following keyboard layout information:

1.) The positions of the keys on the keyboard,

2.) the assigned characters to the keys,

3.) for each device orientation and for the two relevant input types.

As previously specified the positions of the keys and the corresponding characters are referred to as attack layout. Android does not provide an API call to retrieve layout information. We therefore implemented two approaches, *ResourceProvider* and *DynamicProvider*, to solve the challenge **C1**.

Each individual provider, e.g. *ResourceProvider* or *DynamicProvider*, needs to implement a method called `getEntryKeyboard(Height)`. The layout returned by this function is the main page for the currently enabled keyboard. The method also takes the keyboard type into consideration, therefore the height is passed as an additional parameter to this function. The height can be used to infer the keyboard type as described in Section 6.3.2. The actual inference process was delegated to the layout providers as they require the height as well. For the case when the main attack layout is drawn and the user clicks on a key to switch between layout pages, the interface must provide an additional function where the following layouts can be retrieved via the method `getKeyboard(Key)` that takes the clicked key as an argument. If the passed key is used to switch between layout pages, the function should return the page corresponding to the clicked key. Using these two functions, each layout provider can be used regardless of their implementation due to its uniform structure.

6.5.1. Layout Definition

To provide a keyboard attack layout, we first need to settle on one definition to only have one interpretation. We encode a keyboard layout (with all its different layout pages) using an object of Android's class `Keyboard`. This object maintains the positions and dimensions of each key and is also able to parse XML resource files of keyboard layouts [65]. Usually, function keys are defined by using integer constants but for clarity, we decided to encode function keys by assigning the following labels:

- `[Space]` – is the `label` of the space bar key
- `[SHIFT]` – is the `label` of the shift key
- `[Del]` – is the `label` of the delete key
- `[Enter]` – is the `label` of the enter key
- `[Switch]` – is the `label` of the key responsible for switching to other layout pages

Pages Android's `Keyboard` class supports multiple definitions in one file to support different layout pages. Each page must be identifiable by a unique value in the `keyboardMode` attribute. Some constants as seen in Table 6.1 were defined for our layout definition. By providing the XML file and the desired mode to the constructor of the `Keyboard` class, a keyboard can be acquired that only consists of the keys from the passed mode. We require that the main layout, which is the entry point, has the mode zero for portrait layout and one for landscape. Concerning PIN layouts, the entry points should have the modes six and seven. All additional layout pages are selected dynamically using `[Switch]` keys. Of course, those follow-up layout pages, such as the symbols page need to be marked as well, using one of the mode constants. As Android's `Keyboard` class does not support the handling of the `[Switch]` key, this feature needs to be implemented. To be able to do so,

Mode name	**Mode ID**
keyboard_normal_portrait	0
keyboard_normal_landscape	1
keyboard_symbol_portrait_pageOne	2
keyboard_symbol_landscape_pageOne	3
keyboard_symbol_portrait_pageTwo	4
keyboard_symbol_landscape_pageTwo	5
keyboard_pin_portrait	6
keyboard_pin_landscape	7
keyboard_symbol_number_portrait	8
keyboard_symbol_number_landscape	9
keyboard_number_portrait	10
keyboard_number_landscape	11

Table 6.1.: Keyboard modes of the layout definition

the `[Switch]` key must provide a mode ID to the following page that is displayed, if the key is clicked. We observed that Android automatically resolves resource IDs defined in XML files to their corresponding value at runtime. However, the `Keyboard` class requests the actual resource ID as argument for the mode [65]. If the attribute `popupKeyboard` is set to a keyboard mode, the attribute `popupResId` automatically obtains the resource ID of the provided mode integer. If a `[Switch]` key is clicked, the resource ID of the mode for the following layout page is therefore stored in the `popupResId` attribute.

We are now able to either manually create a new Keyboard layout by creating a new list of `Key` objects or by loading an XML file by using the constructor of the `Keyboard` class and calling `getKeys()` function to obtain the list of `Key` objects [65]. An example XML attack layout for Google's Gboard is shown in Appendix Listing 10.2.

6.5.2. Resource Layout Provider

The purpose of the *ResourceProvider* is to give an adversary the possibility to create attack layout files prior to the execution of the malicious app. Those attack layouts are saved in the resources of the malicious app. Each app on Android has the ability to define resources. "Resources are the additional files and static content that your code uses, such as bitmaps, layout definitions, user interface strings, animation instructions, and more" [66]. It is intended to act as a fall-back solution if the dynamic approach is not applicable. To select the matching attack layout for the currently active keyboard, an ID for it is necessary.

6.5.2.1. Unique Identifier

In order to select the correct XML attack layout from the malicious app's resources, an identifier is necessary. Our goal is to assemble the Unique IDentifier (UID) in such a way that it changes if the selected keyboard changes (**UID.1**). Additionally, the selected language that is represented by the `InputMethodSubtype` should be considered as well (**UID.2**). We therefore need to determine a function mapping a tuple of `InputMethod-Info` and `InputMethodSubtype` to a UID. The UID should then be used as the filename for the attack layout stored in the resources.

In order to fulfill requirement **UID.1**, we chose to take the `applicationId` of the currently selected `InputMethod` into account. The `applicationId` is guaranteed to be unique among other Android applications [67]. As a result, when the user installs a new keyboard, the UID changes accordingly.

The second part of the UID considers changes in the selected language (**UID.2**). This part therefore considers a unique property of the currently selected `InputMethodSubtype`. The subtype provides an attribute called `getLocale()` that returns a locale character sequence. The locale is a value that identifies a language. However, this method is deprecated as of API level 24 (Android 7.0) [52]. A similar method called `getLanguageTag()` was introduced, it is used to return a BCP-47 language tag. "An BCP-47 language tag is a code to identify human languages. For example the tag *en* stands for English" [68]. We discovered that it is not mandatory to return a value from the `getLanguageTag()` method. Some keyboards, such as the AOSP and Google Gboard keyboard do not provide this tag, therefore another property is necessary to identify the currently selected language. We observed that the value returned from the `hashCode()` method on the respective `InputMethodSubtype` object works as well as, among other things, its recommended for an efficient `hashCode()` implementation that two identical objects always produce same integer results [52]. If the AOSP keyboard is used, this behavior can be verified as shown in Table 6.2.

We tested 74 different locales and achieved unique integer results for the AOSP keyboard. The values remained identical after a reboot of the device. When using Google's Gboard, however, this did not produce distinct values. This is due to the Gboard using the same `InputMethodSubtype` for all languages. In order to still support different languages, the current Android system language is taken into account. The locale of Android can be queried using the `getLocales()` method. A limitation of this approach is that the locale from Android may differ from the selected keyboard language and can therefore produce inaccurate results.

Listing 6.1: Creation of UID for current Keyboard

```
InputMethodManager imm = (InputMethodManager)
    ↪ ctx.getSystemService(Context.INPUT_METHOD_SERVICE);
InputMethodInfo imi = getCurrentInputMethod();
int hashcode = imm.getCurrentInputMethodSubtype().hashCode();
String returnString = imi.getPackageName() +
        Integer.toString(hashcode).replace("-", "_") +
        Configuration.getLocales().get(0).toLowerCase();
return returnString.replace(".", "_");
```

We create the UID based on three values. The first part is the currently selected InputMethod's `applicationId`. The second part consists of the hash code returned by the `InputMethodSubtype`. As the third part, Android's current locale is appended. In order to comply with Android's resource file naming conventions, all letters are converted to lowercase and dots and minus characters are replaced with underscores [69].

6.5.2.2. Layout Retrieval

If the *KeyloggerManager* requests a new layout, the first step is to determine whether or not the keyboard type can be inferred by height. If both keyboard types have the same height, the alphanumeric layout is selected due to the greater attack vector as discussed in Section 6.3.2. If the keyboard type can be inferred reliably, the layout is selected accordingly by providing the matching mode ID constant.

Layout Entry Point The malicious app then uses the calculated UID to search for the predefined XML layout file in the own resources. If a matching layout is found, the execution

Locale	**UID** (`applicationId`, `hashCode()`, `getLocales()[0]`)
en_US	com_android_inputmethod_latin921088104en_us
de	com_android_inputmethod_latin774684257de
ru	com_android_inputmethod_latin1983547218ru
ar	com_android_inputmethod_latin1494081088ar
in	com_android_inputmethod_latin2108597344in
nl	com_android_inputmethod_latin1067440414nl

Table 6.2.: Selected Unique Identifiers for the AOSP keyboard

proceeds by selecting the type and orientation of the layout. Entry points are defined by setting the keyboard mode in the predefined layout file to either *keyboard_*_portrait* or *keyboard_*_landscape*. The matching layout is then parsed using the Android class `Keyboard`. We can simply pass the previously determined resource ID to the constructor and receive a list of keys and their positions using the `getKeys()` function. The `Keyboard` class is deprecated as of API level 29. This, however, is not relevant as the attack is targeting Android 7.0, which corresponds to API level 24 [65]. After obtaining the list of keys, their coordinates need to be translated to the screen bottom. This is necessary as the key's coordinates are arranged relative to the keyboard. We can achieve this by calculating the following equation for each individual key:

$$\texttt{Key}.y := \texttt{Key}.y + \texttt{Display}.\mathit{height} - \texttt{Keyboard}.\mathit{height} - \texttt{status\_bar\_height}$$

The height of the status bar in pixels can be queried from the configuration value `status_bar_height` [70]. After the process of shifting is completed, the layout can be passed to the requesting instance.

Following Layouts All following layout pages are requested by passing the clicked key as an argument to the *ResourceProvider*. The method then validates that the passed key is indeed a switch key by verifying the label `[Switch]`. As defined in Section 6.5.1, each key which purpose is to switch between layout pages is obligated to provide a resource ID for the following mode in the `popupResId` attribute. The layout is retrieved identical to the process described above but uses the passed keyboard mode instead of the entry point.

6.5.3. Dynamic Layout Provider

In this section, we elaborate on the approach to dynamically query the currently selected keyboard layout. We will show how to use Android's methods to access foreign resource files to obtain keyboard layout files.

We observed that Google's Gboard and the AOSP keyboard save their layout files in the app resources. The actual key declarations are stored in various XML files in both apps. However, they do not share the same type definition and are therefore not uniform as seen in the Listings 6.2 and 6.3. As a result, a parser that supports the specifics of the type definition has to be developed for each keyboard individually. The focus was set on the AOSP keyboard as it is a very well documented keyboard and specifications can be extracted without excessive use of reverse engineering. The Gboard, on the other hand, is closed-source and therefore the specifications are not publicly available. Additionally, well-known custom operating systems based on Android such as "Lineage OS" use the AOSP keyboard [71]. The implemented parser for the AOSP keyboard shall act as proof of concept that it is possible to develop parsers for other keyboards as well as their functioning is similar.

Listing 6.2: Google Gboard definition for the letter q [72]

```
<softkey
        id="@+id/softkey_latin_small_letter_q"
        press_data="q"/>
<mapping
        key_id="@+id/softkey_latin_small_letter_q"
        view_id="@+id/key_pos_0_0"/>
```

Listing 6.3: AOSP keyboard definition for the letter q [73]

```
<Key
        latin:keySpec="!text/keyspec_q"
        latin:keyHintLabel="1"
        latin:additionalMoreKeys="1"
        latin:moreKeys="!text/morekeys_q" />
```

The main goal of the *DynamicProvider* is therefore to obtain the layout files from the app resources of the AOSP keyboard. Those layout files are then parsed to dynamically infer the exact attack layout. Hence, the goals are structured as follows:

a) Determine relevant Layout XML files from the AOSP keyboard

b) Obtain relevant layouts from AOSP keyboard's resources

c) Parse the obtained layout files and resolve dependencies

6.5.3.1. AOSP Keyboard Layout Definition

We start by presenting the AOSP keyboard layout type definition. We observed that each AOSP keyboard layout is structured as follows: there is a root document called `keyboard_layout_set_*` whereas the $\star$ defines the keyboard name that is related to the language, for instance: qwerty, hebrew, georgian. The `keyboard_layout_set_*` contains the definitions for single layout pages grouped under `<Element>` tags as seen in Figure 6.5. Each of those tags has the attribute `elementName` that specifies the keyboard type and page. This can, for instance, be:

1. `alphabet` – for the alphanumeric keyboard layout
2. `symbols` – for the symbol page of the alphanumeric keyboard
3. `number` – for the PIN keyboard.

The actual layout definitions are not stored in the tag but are referenced using the `elementKeyboard` attribute. It contains a reference to the file that contains the layout definition. Each `<Element>` tag can be resolved to a file `kbd_*`, e.g., `kbd_qwerty` consisting of a `<Keyboard>` tag as the root element. This tag can have multiple `<Row>` declarations, which represent a keyboard row. In each row, there are several `<Key>` tags, which define the actual keys for the layout. Those tags have various attributes, for example, the attribute `keySpec` defines which character is represented by that key. Both the `<Row>` and `<Key>` tag can define attributes, for instance, `keyWidth`. However, if the attribute is defined inside a `<Row>` tag, it is set at the granularity of rows. That results in each `<Key>`, which is a child of that row, adopting the specified attribute. If the attribute is defined inside the `<Key>` tag, it overrides inherited values.

In order to be able to reuse any layout definition, the AOSP keyboard supports `<include>` tags. Those can be used to add a dependency for another layout file that contains the actual declaration. The declarations can range from single keys to complete rows and are not restricted by any means. Each tag has the attribute `keyboardLayout`, which defines the reference to the layout file that is included. An example of definitions with `<include>` tags is provided in Listings 6.4.

Listing 6.4: AOSP keyboard file rows_qwerty.xml

```
<Row latin:keyWidth="10%p">
        <include latin:keyboardLayout="@xml/rowkeys_qwerty1"/>
</Row>
<Row latin:keyWidth="10%p">
        <Key latin:keySpec="!text/keyspec_q"
                 latin:keyHintLabel="1"
                 latin:additionalMoreKeys="1"
                 latin:moreKeys="!text/morekeys_q" />
        <include latin:keyboardLayout="@xml/rowkeys_qwerty3"/>
</Row>
```

To support even more sophisticated layouts, `<switch>` statements can be included in any XML file. Those are used like conventional switch statements and represent conditional branches inside the nested files. These may look like shown in the example presented in Listing 6.5.

Listing 6.5: An example Switch statement

```
<switch>
        <case latin:passwordInput="true">
                <include latin:keyboardLayout="@xml/
                        rows_number_password" />
        </case>

        <default>
                <include latin:keyboardLayout="@xml/
                        rows_number_normal" />
        </default>
</switch>
```

The root element is a `<switch>` tag. There can be any number of `<case>` statements. Each `<case>` statement has an attribute that acts as condition and needs to be validated. If the condition is satisfied, the instructions inside the `<case>` tag are executed. Each `<switch>` statement must contain a `<default>` branch that is executed in case no condition could be fulfilled [73]. The hierarchy of XML tags is shown in Figure 6.5.

6.5.3.2. Obtain Relevant Keyboard Layout Set

In this section, we describe the process of obtaining relevant layout pages from the AOSP keyboard. The first step is to detect the current keyboard language. The keyboard language should be mapped to a XML file, which is then parsed to an attack layout.

Keyboard Language The first step is to detect the current language of the keyboard. As the amount of keys and their positions may differ for varying scripts, the keyboard script needs to be detected in order to launch a successful attack. The Latin script is the basis of most alphabets [45]. But there are many others like Cyrillic, Arabic or Malayalam script. The AOSP keyboard supports around 80 different languages (ref B.2). Our goal is to support the same languages in order to perfectly mimic the AOSP keyboard. By

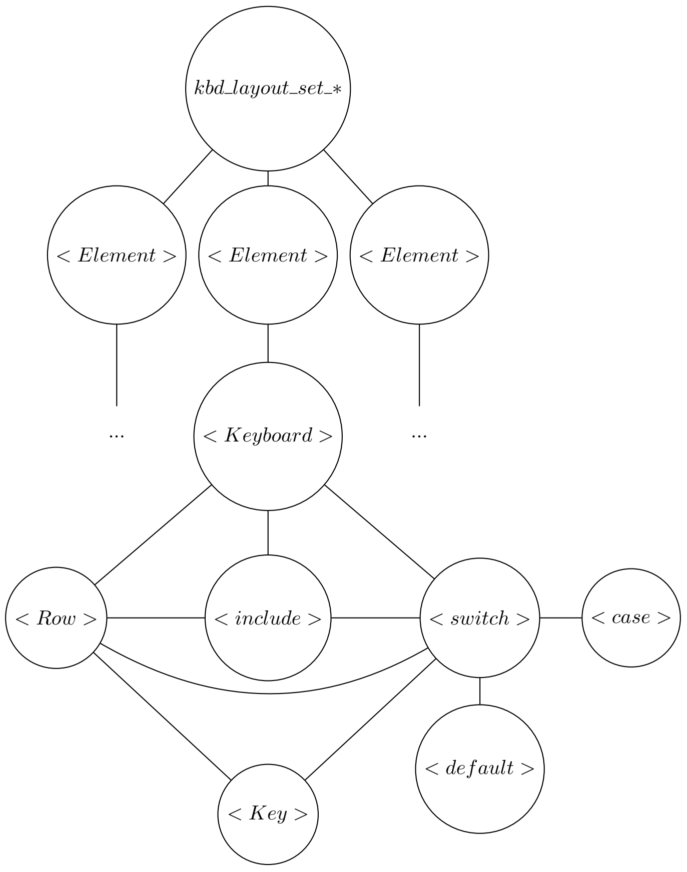


Figure 6.5.: Hierarchy of AOSP XML tags

obtaining the currently selected `InputMethodSubtype`, the locale of it can be obtained. "A locale is a set of parameters that defines the user's language [and] region" [74].

Obtain Resource ID for Keyboard Layout Set We can obtain the package name of the keyboard by calling the `getPackageName()` function from the `InputMethodInfo`. "Package objects contain version information about the implementation and specification of a Java package" [75]. Packages can be identified by their fully-qualified name for example: `uni.wue.cloak.and.dagger`. Android's `PackageManager` provides the `getResourcesForApplication()` method to obtain the resources from any app. The returned `Resources` object provides various methods to access single files or objects. For example, the method `getIdentifier()` is used to return the resource ID. Once the ID is obtained, an adversary can obtain a file by calling the method which matches the type. For example, a String resource can be obtained by calling the `getString(id)` function [76].

By using this technique, the resource ID of a lookup map can be obtained from the AOSP keyboard's resources, that resolves locales to their corresponding keyboard and is called `locale_and_extra_value_to_keyboard_layout_set_map` [77]. The obtained resource ID can then be used to obtain the actual lookup map. By obtaining the entry from the table that corresponds to the keyboard's locale, the resource name of the keyboard layout set can be inferred. The resource name can then be resolved to a resource ID by using the `getIdentifier()` method.

6.5.3.3. Parse Keyboard Layout Set

As the Resource ID for the currently active keyboard layout set was obtained in the previous step, this section describes the actual parsing algorithm. The process applies the

Layout name	**Layout ID**
alphabet	0
symbols	5
symbolsShifted	6
phone	7
phoneSymbols	8
number	9

Table 6.3.: Layout name to layout ID table

following steps:

1.) Select relevant keyboard pages,

2.) Resolve each dependency from `<include>` statements,

3.) Evaluate each `<switch>` statement and act accordingly,

4.) Obtain further values necessary to parse the layout,

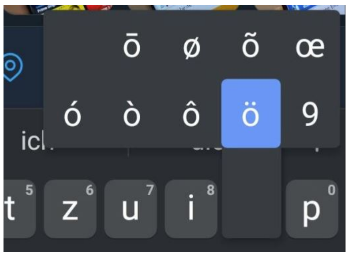
5.) Parse relevant tags.

A `XmlResourceParser` can be obtained from the resource ID in order to access its contents. As mentioned before, the keyboard layout sets contain multiple keyboards and their pages. An adversary is only interested in alphanumeric and numeric passwords. Additionally, we decided to support the layout to enter phone numbers. Internally, the keyboard layout names are represented by integer values. First, we create a lookup table for resolving the layout names to their respective numbers as shown in Table 6.3. In the next step, the resource identifiers for the single layout pages are obtained. For instance, in order to parse the first layout page of the keyboard used for alphanumeric passwords, the `<Element>` tag that has the value zero set for the `latin:elementName` attribute has to be found. If the right `<Element>` was found, the resource ID of this layout can be obtained by accessing the value stored in `latin:elementKeyboard`. This resource identifier can then be parsed in the next step.

Resolve Dependencies Due to `<include>` statements, the layout can contain dependencies to different XML files that need to be resolved. A recursive algorithm should evaluate each tag and obtain the corresponding layout file in case of `<include>` statements. However, not all `<include>` tags need to be parsed in case they are bound by a condition using `<switch>` tags. The algorithm has therefore to be capable of evaluating conditional branches.

Switch statements are mostly used to define additional keys, such as vowel mutations. These can be entered by long-pressing a key (Figure 6.6) and depend on the selected language. Furthermore, they are used to determine if a language switch button is included in the layout in case the user selected multiple languages. We decided to ignore the additional keys as the attack is only able to detect single clicks. Additionally, we assumed in the adversarial model that the user only selected one active keyboard language. As a result, only the default branch of `<switch>` statements is parsed. Concerning PIN password layouts, the branch with the `latin:passwordInput` attribute set to true is parsed due to the fact that PIN password layouts have additional keys called “spacers” that are used for centering the layout [73]. The spacers, marked using `[S]`, are shown in Figure 7.1a.

Figure 6.6.: Gboard vowel mutations [78]



Variable	Meaning	Configuration value
`mTopPadding`	space between content and top border	`config_keyboard_top_padding_holo`
`mBottomPadding`	space between content and bottom border	`config_keyboard_bottom_padding_holo`
`mDefaultKeyWidth`	default key width	`config_more_keys_keyboard_key_width`
`mHorizontalGap`	horizontal gap between two keys	`config_key_horizontal_gap_holo`
`mVerticalGap`	vertical gap between two keys	`config_key_vertical_gap_holo`
`mDefaultRowHeight`	default row height	`config_more_keys_keyboard_key_height`

Table 6.4.: AOSP keyboard configuration values

As the algorithm is now capable of evaluating which tags are relevant, the parsing process can continue. Each `<include>` statement represents a stand-alone XML file. The `latin:keyboardLayout` attribute contains an @ character concatenated with the resource ID. The algorithm can therefore be called recursively with the newly obtained resource XML file. This process is repeated for each dependency.

Configuration Values Not all keys have explicitly set values for their width and height due to the fact that only those differing from the default values must be defined explicitly. All others inherit the default width and height defined as keyboard configuration values. Those can be obtained from the AOSP keyboard resources saved as fractional values. We are interested in the values presented in Table 6.4. We observed that even if clicks are outside the key's area, the key click is still detected by the AOSP keyboard, meaning the gaps are for layout purposes only and do not really change the hit-box dimensions of actual keys. However, they still need to be obtained in order to calculate the row height and the key dimensions as the values are relative to the keyboards dimensions. For instance, the default key height is relative to the row height. The row height, however, is relative to the View from the keyboard in which keys are displayed. Finally, the dimensions of this View are dependent on the defined gaps and paddings [79]. In order to have identical behavior, especially concerning the detection of identical key strokes, we decided to ignore the gaps between the keys. The `mDefaultRowHeight` value is therefore used as default key height and the `mDefaultKeyWidth` value as default width.

Parse Rows and Keys If the recursive algorithm detects `<Row>` declarations, a method to parse them is executed. Only `keyWidth` and `keyHeight` attributes are relevant as they are used to define the dimensions at the granularity of rows. The attributes can be defined by using relative or absolute values. Values that are relative to the keyboard dimensions can be detected by the character sequence *%p* at the end of declarations [73].

Label Name	Keyboard Page Transition
`toSymbolKeyStyle`	alphanumeric to symbol page one
`backFromMoreSymbolKeyStyle`	from symbol page two to symbol page one
`toMoreSymbolKeyStyle`	from symbol page one to symbol page two
`toAlphaKeyStyle`	from symbol page one to alphanumeric
`numPhoneToSymbolKeyStyle`	from number to number symbol
`numPhoneToNumericKeyStyle`	from number symbol to number

Table 6.5.: Switch key labels from the AOSP Definition used to switch between layout pages [73]

If the recursive algorithm finds a `<Key>` declaration, the `keyWidth` and `keyHeight` attributes are parsed as well, as long as they are present. As mentioned above, attributes defined for single keys override attributes defined for rows. Additionally, the *keySpec* attribute is parsed in order to obtain the key's label. Some keys have no `keySpec` attribute but a `keyStyle` defined, for instance, `lessKeyStyle`, `deleteKeyStyle` or `enterKeyStyle` [73]. As the values are unique, we decided to adopt them as key labels. However, for aesthetic reasons, some characters are replaced by their actual symbol which can be seen in Appendix Table B.1.

The algorithm has to convert functional keys, especially the key used to switch between layout pages, to our used definition. As a result, the AOSP definition of a switch key has to be converted to our own definition. Concerning the AOSP keyboard, switch keys are defined by setting the `latin:keyStyle` attribute to a value like `toSymbolKeyStyle` or `backFromMoreSymbolKeyStyle`. Those label names identify a keyboard page transition, for instance, from the alphanumeric keyboard to page one of the symbol keyboard. The `latin:keyStyle` attribute is then evaluated and the target layout page for the `[Switch]` key is defined by the label name from Table 6.5.

Center layout The AOSP keyboard centers all keys in each row as seen, for instance, in Figure 2.2a if no spacers are defined. The process is therefore adopted and each key is centered in its respective row by expanding the width of left and right most key to equal parts until no space is left. We observed that the expanded keys have a larger area for which clicks are detected (hit-box) due to their greater size. However, the AOSP keyboard shows identical behavior.

6.5.4. Layout Learning

The dynamic attack layout learning approach was not implemented as its performance was not good enough. Yet we introduce it briefly as it was considered to be implemented. As the app is able to create an input field in which the user is lured to type some keys, the app can check which text was typed in the `EditText`. Previously, an adversary has drawn a very narrow grid on top of the keyboard. This enables an adversary to infer which grid field equals which keystroke and therefore dynamically map each grid field to a keyboard key. This approach, however, is quite slow and requires manual user interaction but theoretically can be improved by using statistical predictive modeling approaches, such as decision trees [80]. The approach only works in case the user actively clicks keys on the keyboard, hence, user interaction is required. The approach could be autonomous by emulating the clicks programmatically. This, however, needs the `INJECT_EVENTS` permission. This is a so-called signature permission defined by the OS and can therefore only be granted to apps that share the same private key as the operating system [81]. If we were able to obtain this permission, it would enable us to automatically learn the keyboard layout without being dependent on any user interaction. Additionally, as elaborated in Section 5.2, the drawing of narrow grid fields produces very large resource overheads and tends to have a bad performance, which makes this approach hardly feasible.

6.6. Keylogger Manager

In this section, the implementation of the *KeyloggerManager* module is presented. The goal is to perfectly place the “Invisible Grid” attack on a device. We, therefore, have to combine information and approaches from the previous modules. The approach includes the following steps:

a) Get the size of the keyboard using *KeyboardSizeProvider*,

b) Get the attack layout from matching Layout Provider e.g. *ResourceProvider* or *DynamicProvider*,

c) Detect the state of the keyboard using *KeyboardPresenceProvider*,

d) Utilize “Invisible Grid” attack to infer keystrokes.

Initialization Stage The initialization step applies methods from the *KeyboardSizeProvider* to deliberately open the keyboard in order to take measurements. Since the *DynamicProvider* only supports the AOSP keyboard it is only used in case the AOSP keyboard is actually enabled. For all other keyboards, the *ResourceProvider* is used, however, it can only be utilized in case the matching attack layout was created prior to app execution. In our implementation, we only created an attack layout for Google’s Gboard English layout. As elaborated in Section 6.5.2 more keyboards can easily be supported by creating their XML attack layout. The measurements from the *KeyboardSizeProvider* are used in the next step to gather the main layout page from the *DynamicProvider* if the AOSP keyboard is used. Otherwise, the *ResourceProvider* is used to gather the layout if it was created prior to app execution. Orientation information necessary to select the layout is gathered from the *KeyboardPresenceProvider.*

6.6.1. Placement Stage

After the initialization step is finished, the actual attack is placed. The goal is to combine the modules to infer the following information:

1.) Detect if relevant clicks occurred inside the keyboards key area,

2.) Detect keyboard state changes to be able to reset layout pages.

Relevant Clicks The basic attack strategy is structured as follows. In the first step, only relevant clicks are evaluated. Clicks that have occurred outside the keyboards key region can not reach a key and do not need to be evaluated. The app is therefore supposed to only evaluate clicks that occurred inside the key region and detect for those if the keyboard state is *Opened.* This distinction is necessary as the `ExtractUi` mode has a key region and a text region as seen in Figure 2.1c. We can infer if a click occurred inside the key region by applying the following approach, which will ultimately be combined with the *Advanced Window Method.*

It is assumed we draw the overlay O_2 in the top left corner with a width and height of zero and the pass-through flags set. Additionally, the flag `FLAG_WATCH_OUTSIDE_-TOUCH` is set. As the overlay inherits pass-through behavior, each click is considered outside and therefore each click will trigger the outside touch event. In the next step, the keyboard overlays O_3, O_4, ..., O_n are drawn. The positions of the keys are provided by a

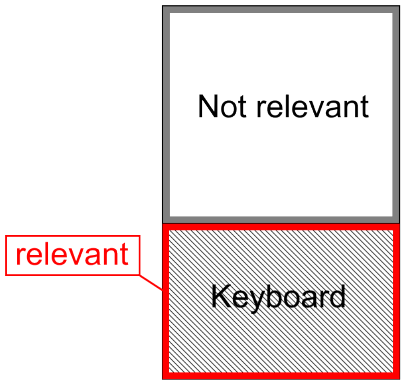


Figure 6.7.: Clicks that have occurred outside the keyboards key region are not considered relevant.

KeyboardLayoutProvider and therefore each overlay has pairwise disjoint x-y-projections. As the overlays were added in ascending order, we can assume that the z-index of O_2 is smaller than the z-index of O_j for all $j > 2$. All overlays are therefore ordered by increasing z-index and have pairwise disjoint x-y-projections.

Based on the definitions presented in Section 4, the following detection scheme can be used: Assuming $p \notin O_i$ for all $i \in \mathbb{N}$, we have $O_2.\texttt{OBSCURED} = 0$ as $p \notin O_j$ for all $j > 2$ with $j \in \mathbb{N}$. Conversely, if $p \in O_j$ with $j > 2$ we have, by definition, $O_2.\texttt{OBSCURED} = 1$.

Simplified that means: if a click is detected by O_2 and $O_2.\texttt{OBSCURED} = 0$, the click must have occurred outside the keys region. Otherwise, an overlay in the keys area was clicked and therefore overlay O_2 is obscured.

This detection scheme enables the malicious app to infer which clicks are relevant. For those considered relevant due to the condition $O_2.\texttt{OBSCURED} = 1$ being fulfilled, the current keyboard state is evaluated by combining this detection technique with the *Advanced Window Method*. A specially-crafted overlay O_1 is therefore drawn prior to all other overlays. If overlay O_2 receives an outside touch event and the click is considered relevant, the malicious app waits for a small period of time and validates if O_1 has received an event as well. By using this approach, only relevant clicks are evaluated and the keyboard state is taken into consideration as well.

Keyboard State Changes Keyboard state changes need to be detected in order to reset layout pages. Due to this requirement, the *Window Method* is utilized. If the state of the keyboard is set to "Closed", the layout page is reset to the entry point layout page. Additionally, the `[Shift]` key is reset to disabled as well.

As Herley et al. discovered, a limitation of classical keyloggers is that they are not able to eavesdrop on sensitive input correctly if the user changes cursor positions in the `EditText` [82]. We can't bypass this limitation but the impact can be lowered. Therefore a feature was implemented that exploits the way Android keyboards work. If a user clicks outside an `EditText's` area, the `EditText` that is currently focused will lose focus. If no `EditText` is focused, the Android keyboard will automatically close.

The state of the keyboard is detected by the *Window Method* presented in Section 6.3.1. Each click that is not considered relevant, meaning $O_2.\texttt{OBSCURED} = 0$, is outside of the region of the keyboard. For these clicks, the malicious app is able to check if the keyboard closed after 500 milliseconds. If the keyboard did not close, the `EditText` still has focus and therefore the probability is very high that the user changed the cursor position. In this case, a warning is printed to the output that results may be inaccurate.

Implementation Details As we have already elaborated, drawing a specially-crafted overlay will result in all overlays, currently on the screen, being shifted below the keyboard in terms of their z-index. Due to this circumstance, the drawing order is of severe importance. An attack would not be successful if the keyboard overlays are behind the keyboard, as clicks would not be detected anymore. That means all overlays, which overlay keys on the keyboard, have to be drawn last to not be shifted. The *Window Method* must therefore be added first since it contains a specially-crafted overlay. Afterward, for the same reason, the *Advanced Window Method* is applied. As the last step, the single keyboard overlays are drawn.

For the implementation of the *Advanced Window Method*, combined with the detection of relevant clicks, concurrent programming is necessary. Due to the overlay O_1 being added first by the *Advanced Window Method*, it inherits the lowest z-index. Therefore, it is notified last in case an outside click occurs. In case overlay O_2 detects an outside key click, a new thread is started that waits for ten milliseconds to get a signal from the overlay O_1. The signal feature is implemented using a synchronized object that is used to safely access objects from multiple threads [83]. The signal is provided in case the overlay O_1 receives a `FLAG_WATCH_OUTSIDE_TOUCH` event. The thread then checks if the signal was set or if a timeout occurred. In case of a timeout, the malicious app can infer that the keyboard is opened, otherwise, a `FLAG_WATCH_OUTSIDE_TOUCH` event would have been fired for the overlay O_1. However, if the overlay O_1 does receive a `FLAG_WATCH_OUTSIDE_TOUCH` event, the malicious app can infer that the keyboard is not opened in any of the three keyboard modes.

6.6.2. Attack Stage

Each keyboard overlay O_3, O_4, ..., O_n is stored in the `keyViews` array. The `keyViews` were added last and therefore have higher z-indices than all other overlays. In case the overlay O_2 detects a click, the *Advanced Window Method* determines if the click occurred inside the keyboards key area, i.e., is relevant. Additionally, since the `FLAG_WATCH_OUTSIDE_TOUCH` event order is descending, the event was processed by every overlay in the `keyView` array. Each time a `keyView` receives a callback from the `View.OnTouchListener` due to a `FLAG_WATCH_OUTSIDE_TOUCH` event, it validates whether or not the `OBSCURED` flag is set for it. If it is obscured, an integer called `counter` is incremented.

If O_2 detects an obscured outside click, i.e., a relevant click, the amount of obscured overlays from the *keyViews* array is now stored in the `counter` variable. The clicked overlay can therefore be inferred by using the counter as an index for the `keyViews` array. At the end of each event from the O_2 overlay, the `counter` is reset to zero in order to be able to infer the next key click. In order to be able to infer which key was clicked, a lookup map is created that assigns each `View` the corresponding keyboard `Key`. If a `[Switch]` key is clicked, the method for drawing the layout is called and the `[Switch]` key is passed as an argument to the `getKeyboard(Key)` method from the *KeyboardLayoutProvider* in order to obtain the next layout page.

Listing 6.6: Pseudo Code to infer key strokes

```
keyViews = [$O_3, O_4, ..., O_n$]
counter = 0
clickedKeyView = null
//When executed clickedKeyView contains the clicked key

outsideTouchEventCallback(
new function($O$)
if $O$ is $O_2$
if $O$.OBSCURED
if $O_1$.SIGNAL
clickedKeyView = null
else if $O_1$.TIMEOUT
clickedKeyView = keyViews[counter]
else
clickedKeyView = null
if keyboard does not close after 500ms print warning

counter = 0
else s.t. $O$ equal to $O_3, O_4, ..., O_n$
if $O$.OBSCURED
counter++
)
```

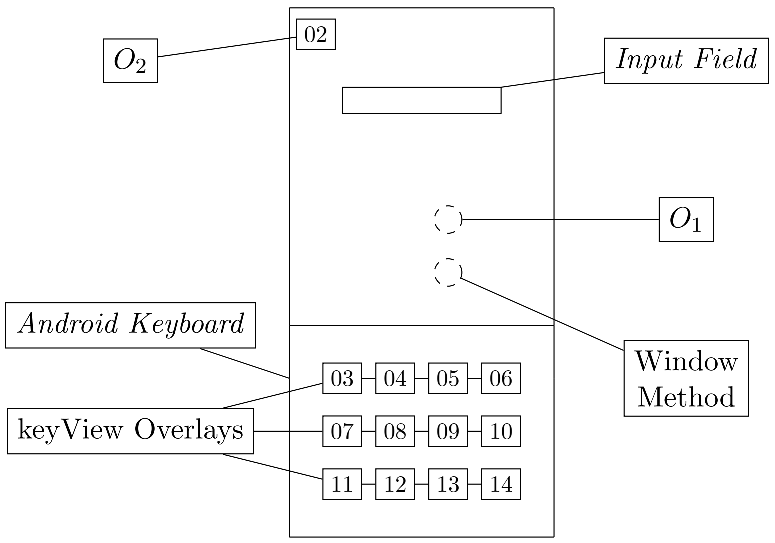


Figure 6.8.: *KeyloggerManager* attack placement (numeric values inside boxes represent z-indices). Circles represent that the overlay has a lower z-index than the keyboard.

Drawing the Attack As shown in Section 5.3, the drawing of overlays takes a lot of time. Therefore, the overlays are kept on the screen and in case of layout changes the characters and positions are updated. If the new layout requires a different amount of overlays than the previous one, the numerical difference Δ is calculated. Hence, Δ overlays are either added or removed. Afterward new layout parameters (`WindowManager.LayoutParams`) with the updated attributes are created. By calling the `updateViewLayout()` method from the `WindowManager`, the new layouts are assigned [36].

Sensitive and Non-Sensitive Input The height of the keyboard for different input types was measured using the *KeyboardSizeProvider* in the initialization stage. The app is therefore able to distinguish between sensitive and non-sensitive input if there is an injective mapping from (measured) keyboard height to keyboard type. Concerning the AOSP keyboard, we observed that the `TYPE_TEXT` input usually spawns a suggestions bar on top of the AOSP keyboard in order to provide suggestions for the typed input. This increases the height of a keyboard and therefore it is distinguishable from sensitive input fields that usually do not have a suggestion bar. The malicious app prints information if a PIN, alphanumeric password or text input field was detected. However, in case of the AOSP keyboard, the measured heights for PIN and alphanumeric passwords are the same. Hence, the app is only able to distinguish between passwords and normal text.

6.7. Logger

In the following paragraphs, the module *KeyloggerLogger* is presented. Its purpose is to handle all key clicks and print out the characters. In order to infer the correct characters the `[Shift]` key must be handled correctly. The following paragraphs describe the approach and resulting limitations.

Each key click is forwarded to the *KeyloggerLogger* module by calling the `keyClicked(Key)` method and passing the clicked `Key` as an argument. In the next step, the method tries to infer if the clicked key is upper- or lowercase.

Distinguishing Upper- and Lowercase This is achieved by evaluating clicks on shift keys. Concerning the AOSP keyboard, we observed that there are three ways to enable shift keys. The first method is to single click on the `[Shift]` key and therefore shift only the next letter. The second way is to click the `[Shift]` key twice in a small period of time that enables caps lock. Caps lock permanently enables the `[Shift]` key until it is clicked again. This function can also be enabled by long clicking the `[Shift]` key, that represents the third way.

Single clicks on the `[Shift]` key can be detected trivially. Additionally, the malicious app stores the timestamp of each `[Shift]` key click in order to infer if a subsequent click took place. If the two successive clicks are in a time span of 300 milliseconds, caps lock is enabled. The threshold was adopted from the AOSP keyboard, due to the observation that caps lock is enabled for a time span of 300 milliseconds but not for 305 milliseconds.

The third way of enabling the `[Shift]` key, by long clicking the key, cannot be implemented due to the functioning of the attack. The limitation is that, by design, the malicious app is only able to detect the initial gesture that triggered the `ACTION_OUTSIDE` event. It therefore only gets the "down event" but not the following "up event". As a result, long presses on keys are not detected. The third method for enabling the `[Shift]` key is therefore not implemented.

7. Evaluation

In this chapter, we evaluate the implemented attack by use-case testing it on different devices. A representative sample of the languages supported by the AOSP keyboard as well as Google's Gboard English layout is tested. Furthermore, different device resolutions and orientations are considered.

Android Version	Implementation Working	Additional modification necessary
4.4	✓	✓
5.1	✓	×
6.0	(✓)	✓
7.0	✓	×
7.1	✓	✓
Greater 7.1	×	×

Table 7.1.: Overview of use-case tested implementation for different Android versions. The (✓) indicates that the general attack works on this version but the developed application does not.

7.1. Vulnerable Android Versions

Our implementation targets Android 7.0. However, we evaluate on which other Android versions it works as well. We test the end-to-end attack implementation for the AOSP keyboard. As the AOSP keyboard supports around 80 different languages we only test a representative sample of languages. Additionally, we test the attack for Google's Gboard English layout. We summarize the findings into categories below 7.1 and above 7.1. In the figures 7.1a, 7.1b, 7.1c and 7.1d, we present visible attack overlays for the AOSP keyboard using the *DynamicProvider* and Google's Gboard by utilizing the *ResourceProvider*.

7.1.1. Android 7.1 and below

In this section, we evaluate the implementation for Android versions below 7.0 by use-case testing it on devices that utilize different Android versions. The use-case test was conducted by comparing recorded keystrokes with actual keystrokes for random user input. The random keystrokes made sure that all keyboard pages were utilized. Moreover, upper-

and lowercase letters were tested. Furthermore, the use-case tests were carried out using a representative sample consisting of multiple languages including English, Russian, German, Tamil and Georgian. Additionally, the attack was tested on different screen sizes to guarantee that it is independent of device resolution.

Android 4.4 The implementation works for emulated Android 4.4 devices as long as the app is in the foreground. As soon as the app is sent to the background, all overlays are closed due to the fact that a valid `Context` is necessary to display overlays. The `Context` is destroyed as soon as the app is in the background. This behavior can be circumvented by creating a `Service` that keeps running in the background. In order to create a `Service`, no additional permissions are necessary for Android 4.4 [84]. The attack can run successfully on devices using this Android version by applying this modification.

Android 5.1 The implementation was tested on an emulated Android 5.1 device with a screen resolution of 1080 by 1920 pixels. The security patch level was dated as of 1st November 2015. The implemented attack worked without any restrictions.

Android 6.0 The attack was also tested on an emulated Android 6.0 device with a screen resolution of 480 by 800 pixels. The screen resolution was drastically reduced to determine if the attack is capable of handling different resolutions. The security patch level was dated as of 6th September 2016. The implementation was working smoothly as long as the app is in the foreground. If the app enters the background, the `ACTION_OUTSIDE` events were dropped by Android's `ViewRootImpl`. This method checks if the receiving view has the focus in case it does not, the event is dropped [85].

Due to the dropping of events, the implementation does not work for this version. If `ACTION_OUTSIDE` events are dropped, the counter, which is incremented in case the View is obscured, does not work properly. We are, therefore, not able to infer which grid field was clicked. Zheng et al. [27] managed to successfully launch the attack on Android 6.0 using `TYPE_TOAST` overlays instead of `TYPE_SYSTEM_ALERT`. Nevertheless, we could not reproduce the success by changing the overlay type of our implementation as well.

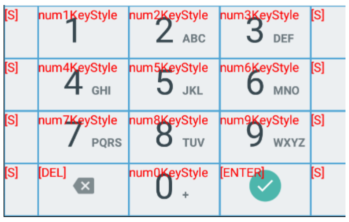
Android 7.0 The attack was use-case tested on an emulated Android 7.0 device with the security patch dated as of 5th June 2017. The attack was tested with the AOSP keyboard for different languages. Google's Gboard was tested as well but only for the English keyboard layout. All three keyboard modes were tested. Furthermore, both device orientations were tested. In all use-case tests, the implementation was able to successfully eavesdrop on sensitive input for different languages. Therefore, we conclude that all implementation challenges were solved for the targeted Android version.

Android 7.1 Android 7.1 introduced two new mitigation techniques to prevent the "Invisible Grid" attack. The first one is a timeout for `TYPE_TOAST` overlays that ensures they can only be shown for a maximum period of 3.5 seconds. The second mitigation is that only one `TYPE_TOAST` overlay per application is allowed [27].

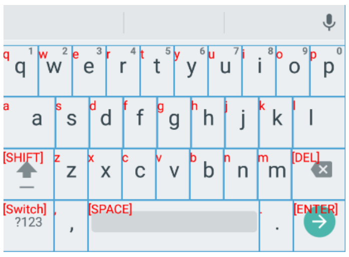
As a result of this behavior, the *Window Method*, presented in Section 6.3.1, no longer works. In order to circumvent this restriction, however, an adversary can use the *Reflection Method*, presented in Section 6.3.1, instead as it does not rely on overlays at all. The implementation appears to be feasible by using this strategy for this version. We conclude that the attack can run successfully on devices by implementing the modifications. Zheng et al. [27] managed to make the attack itself feasible for Android 7.1 devices, which strengthens our hypothesis.

7.1.2. Above Android 7.1

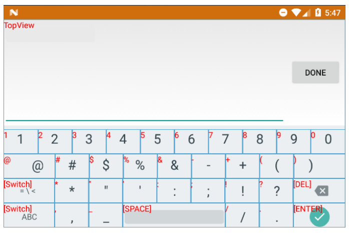
The `TYPE_SYSTEM_ALERT` flag is deprecated as of Android 8.0 [54]. This implies that the implementation does not work for versions greater 7.1. Zheng et al. presented in their



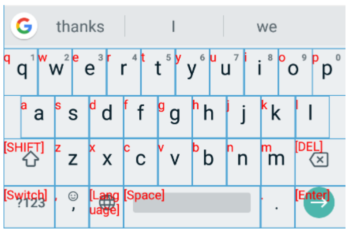
(a) AOSP PIN keyboard with visible attack overlays



(b) AOSP keyboard with visible attack overlays



(c) AOSP keyboard with visible attack overlays in landscape orientation `ExtractUi` mode



(d) Google Gboard keyboard with visible attack overlays

Figure 7.1.: Visible attack overlays for different keyboards.

paper “Android Toast Overlay Attack: ‘Cloak and Dagger’ with No Permissions” [27] an implementation without the use of `TYPE_SYSTEM_ALERT` overlays. Instead, the authors used `TYPE_TOAST` overlays that are not deprecated in Android 8.0.

Due to the restrictions introduced in Android 7.1, only one overlay at a time can be drawn [27]. This prevents the attack as multiple overlays are necessary to infer key strokes.

7.2. Limitations

In this section, we present the limitations of the implemented end-to-end attack. We elaborate on floating keyboards, unsupported swipe gestures, and explain further limitations of the developed approaches.

7.2.1. Supported Keyboards

The first limitation is that only rectangle-shaped keyboards were considered. The attack therefore only works on these. Android itself does not limit the development to rectangle-shaped keyboards; in fact, a keyboard can be implemented in many ways. The most common keyboards, however, are rectangle-shaped and therefore the attack is applicable to a majority of keyboards. The second limitation is that the attack, by design, only works for software keyboards. If users connect a hardware keyboard to a device, an attack is no longer feasible. Floating keyboards that are not placed at the bottom of the screen and can be placed individually by drag-and-drop are also not supported.

7.2.2. Unsupported Swipe Gestures

Another limitation is that the app is only able to detect the initial gesture that triggered the `ACTION_OUTSIDE` event. As a result, it only gets the “down event” but not the following “up event”. It is therefore possible that a user can press a key and slide to another key. This will log the wrong key since the internal keyboard will detect the pressed key using the “up event”. The malicious app, however, is using the “down event” and has no way of accessing the “up event”. While in such a case sensitive data cannot be recovered precisely, an attacker can reduce the search space for his brute-force attack by concentrating on neighboring keys like vowel mutations (as shown in Figure 6.6).

The AOSP keyboard disables swiping at password input by default. Furthermore, swipe keyboards itself pose little limitation since passwords are usually not typed by using “swipe techniques”. This is mainly due to swipe keyboards correlating the swipe path with a word search engine and passwords are often random, hence not found by a word search engine [86]. However, if passphrases consist of natural words, “swipe techniques” may indeed be utilized, which would make the eavesdropping of these not feasible.

7.2.3. Limitations of Keyboard Language Detection

As already mentioned in Section 6.5.2.1, keyboard language detection does not work for some keyboards, such as Google’s Gboard. As a fall-back solution, the system language is assumed to be identical to the keyboard’s language. This assumption is correct for the majority of users. Meanwhile, multiple-selected languages are not supported, e.g. English and German being enabled at the same time. By generalizing our approach, the implementation of this feature appears to be feasible at least for the AOSP keyboard.

7.2.4. Limitations of Advanced Window Method

The *Advanced Window Method* presented in Section 6.3.1 is used to determine keyboard presence for all three modes. Due to the design of this approach, the closing of the keyboard can only be detected by evaluating user clicks. If the user does not click anywhere, the closing process of the landscape `ExtractUi` mode is not detected and therefore the current layout page will not be reset. This limitation only applies to very limited use-cases, such as apps that automatically focus on input fields without the necessity of user interaction and therefore do not impact the feasibility of an attack.

We tested the attack implementation on a variety of versions and have shown that it successfully runs on the targeted platform Android 7.0 and 5.1 without additional modifications. By applying the proposed modifications the attack can run on Android versions 4.4 and 7.1. Furthermore, we have shown that the attack works independently of device resolution and orientation. By testing a representative sample of the around 80 languages supported by the AOSP keyboard we conclude that the implementation supports all the languages as well. Moreover, we have shown that the utilization of custom attack layouts created prior to app execution, for unsupported keyboards such as Goggle’s Gboard, is working as well.

8. Defense Method OverlayShifter

In this section, we present `OverlayShifter`, a novel defense technique preventing the "Keystroke Inference #3" attack from the "Cloak and Dagger" paper by Fratantonio et al. [1]. The defense technique works system-wide and is independent of OS modification while fully preserving usability and overlay functionality. In particular, we first provide theoretical foundations and then present our design and implementation.

8.1. Theoretical Foundations

Consider the adversarial application A and the defender application B. Both applications have distinct package names, hence are completely separated by the Android OS. The adversarial application A is performing the "Keystroke Inference #3" attack. The defender's application B now adds a specially-crafted overlay to the screen. As we elaborated in Section 6.3.1, the so-called specially-crafted overlay will shift all overlays that are currently on the screen behind the keyboard. Hence, the z-indices of all adversarial overlays will be shifted in such a way that they are no longer on top of the keyboard. As we have elaborated in Section 6.3.1, if overlays are behind the keyboard in terms of their z-indices they will no longer receive motion events if input is being entered. That entails, all adversarial overlays from application A will no longer receive motion events in case input is being entered and are therefore no longer able to perform the "Keystroke Inference #3" attack.

We observed that the adversarial overlays are shifted for as long as the specially-crafted overlay from application B is on the screen. However, if A removes the adversarial overlays and adds them for a second time they will be on top of the keyboard again, regardless of the existence of a specially-crafted overlay. In order to circumvent possible re-adding of adversarial overlays, the defender application B needs to continuously add specially-crafted overlays. However, the old overlays don't need to be kept on the screen. Hence, the defender application B can add a new specially-crafted overlay and remove the previous one.

Consider A continuously adds n malicious overlays. The time which is necessary to place n operational adversarial overlays will be defined by the cycle time c_1. Hence, c_1 specifies the time necessary till the last overlay is fully operational, whereas operational means it is placed on the screen and capable of receiving motion events.

$$c_1 = \text{TimestampLastOverlayOperational} - \text{TimestampStartAdding}$$

If the defender application B now continuously adds specially-crafted overlays with a cycle time c_2 and achieves a cycle time such that $c_2 < c_1$, the adversarial process of adding n overlays on top of the keyboard will be disrupted, as the overlays added by A will be shifted behind the keyboard by B.

We have observed that an overlay is only fully operational (able to receive motion events), if the second `onLayoutChange()` event fired. Therefore, we were able to measure the time necessary to place n operational overlays. This was done by storing the timestamp of the second `onLayoutChange()` event for all placed overlays. Additionally, the starting timestamp from which the overlays were initially added was stored. Using those two timestamps we were able to calculate the cycle time c_1. We have conducted the measurement using 30 overlays ($n = 30$) as we have elaborated in Section 5.2 this is the lower bound necessary to perform the attack. We took cycle measurements on a population size of 130 to get reliable results. As seen in Figure 8.1 the minimum adversarial cycle time c_1 is 364 milliseconds. Hence, if the defender achieves a cycle time $c_2 < 364$ the attack is prevented. The smaller cycle time is realistic, as the defender only needs to add one single overlay compared to 30 malicious overlays. Yet, we still measured the time necessary to add one specially-crafted overlay until its operational. The population size was 195 and the maximum defender cycle time c_2 is 62, therefore the condition $c_2 < c_1$ can easily be fulfilled.

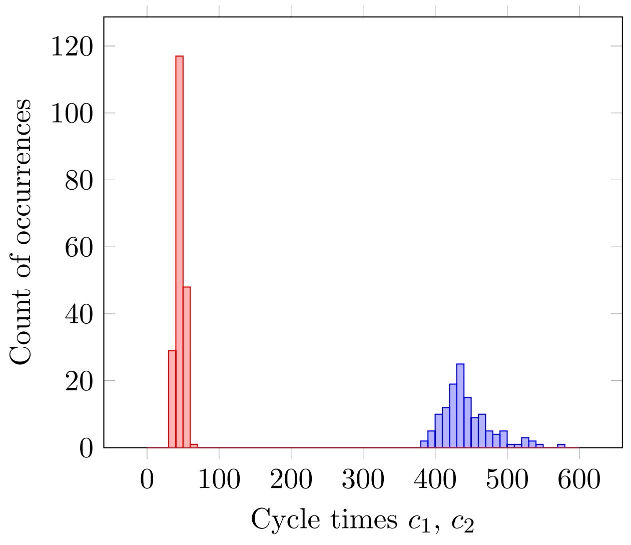


Figure 8.1.: Comparison of adversary (c_1 marked blue) and defender cycle times (c_2 marked red).

8.2. Design and Implementation

`OverlayShifter` is designed to prevent the “Keystroke Inference #3” attack. We have elaborated in the previous section, that the defense technique continuously adds specially-crafted overlays with a lower cycle-time than the adversary. The old overlay can be removed as the new one is being added. We have decided to rotate four specially crafted overlays in order to guarantee that at least one overlay is always present on the screen. This is done by storing four specially-crafted overlays O_0, O_1, O_2, O_3 into an array. In case the overlay O_i is added, the overlay O_j is removed where $j = i - 2$ for $i > 1$ or $j = i + 2$ for $i < 2$ with $i, j \in \mathbb{N}_0$. This rotation of specially-crafted overlays is done in a loop. The process of adding a specially-crafted overlay takes around 44 milliseconds (Median). Hence, we can delay our loop by $t = c_1 - c_2$ which results in $t = 320$. Hence, the function `shiftAllOverlays(int overlayID)` (cf. Appendix C for implementation details) must be called every 319 milliseconds to achieve a defender cycle time $c_2 = 363 < 364 = c_1$. That results in roughly three overlays being placed per second.

The added specially-crafted overlays have a width and height of zero and are therefore invisible to the user. The developed approach can be deployed system-wide which means it does not have to be implemented by each application individually on the Android device. However, due to performance reasons elaborated further on the defense method should only be applied in case sensitive input is expected. `OverlayShifter` works using only the `SYSTEM_ALERT_WINDOW` permission. However, in the current implementation, it is not guaranteed that the `OverlayShifter` continues to work in the background. Hence, additional permissions such as `FOREGROUND_SERVICE` to register a foreground service may be necessary for advanced versions. Foreground services enable continuous background execution on the device. The developed approach does not impair legitimate overlays, as they are most likely not dependent on being above the keyboard.

Future Improvements We have only presented a very basic implementation of the `OverlayShifter` defense method. Further improvements could focus on the performance of the application as loops are usually resource intensive. The performance was not directly measured as we considered this out of scope. However, we were not able to notice any limitations during the test of the `OverlayShifter`. The resource utilization could be further reduced by only activating the defense method in case the keyboard is shown or sensitive input is being entered. The presence of the keyboard can be detected system-wide using any of the three developed approaches in Section 6.3. It should be investigated if other related attacks such as “Hoover” [28] could be prevented as well. In theory, shifting the overlays utilized by this attack behind the keyboard leads to the overlays no longer being able to receive “post-tap hover events”. Hence, the attack could be prevented if the defense method is able to shift the overlays in time. The feasibility still needs to be investigated.

9. Discussion on Countermeasures

In this section we discuss countermeasures, apart from the `OverlayShifter`, such as in-app keyboards. Furthermore, we present characteristics of the attack which facilitate the development of countermeasures.

Security Patching The most effective defense mechanism against the "Invisible Grid" attack is to install the fix for CVE if available. Additionally, `SYSTEM_ALERT_WINDOW` permission should only be granted to trusted applications. Otherwise, malicious activities may be employed. As elaborated previously, many manufacturers no longer support their devices, which therefore do not receive security fixes. In this case, other defense methods that do not rely on modification of the operating system must be applied.

In-app Keyboards One proposed defense mechanism is the use of custom in-app keyboards. Our implementation could be adapted to in-app keyboards by using the in-app keyboard layout as a grid. While this allows the implementation to read the input into the in-app keyboard *when open*, it cannot distinguish typed input from clicks in the bottom area of the screen that do not belong to a keyboard. However, an adversary may still be able to detect the password as it differs from random input and can thus be detected. The attack is even more feasible if the user types the same password again. This is very likely if the attack is running for a sufficiently long time. The approach is similar to a Kasiski & Babbage test [87]. An adversary may be able to detect two identical n-grams with a distance of t. For instance: aK**Password**AGOnCbTQn**Password**BTzgRO.

It may be hard to distinguish normal words from a password. In theory, however, the attack is not prevented just by using in-app keyboards without additional security features. An adversary may get enough hints from analyzing the text to infer sensitive input. Therefore, it is of great importance that the in-app keyboard is also able to detect overlay-based attacks. The detection method could be designed in a way that it is able to detect overlays, for instance by evaluating the `OBSCURED` flag. If the user tries to type on the keyboard and an overlay is detected, it prevents users from typing and displays a warning stating that there might be an ongoing attack. In-app keyboards that detect overlays are, therefore, a fully working defense method. However, the usage of in-app keyboards can reduce user experience as it would be difficult to get used to different keyboards for each app. Another drawback is that legitimate apps can utilize overlays and therefore trigger the obscured flag (e.g. [88]) resulting in false positives.

A huge advantage of in-app keyboards is the low resource overhead. Compared to the `OverlayShifter` the resource utilization is very low, as no loops are required. It should, therefore, be considered to utilize in-app keyboards instead of the `OverlayShifter` in case the reduction in user experience is affordable.

Intrusion Detection Intrusion detection is the utilization of applications that are able to detect, for instance, overlay-based malware. As the malicious app was developed for this approach, we will introduce it briefly. The main goal is to detect malware by collecting traces from ongoing attacks. Due to the variety of possible types of attacks, static analysis can only limitedly be applied in this case. Traces should therefore be analyzed in combination with machine learning techniques to detect attacks. A lot of research effort has to be conducted in this field. However, the utilization of machine learning techniques could just bring the desired success.

In the following, we are going to discuss the characteristics of the attack that facilitate its detection. It is not clear if those characteristics can be detected without OS access. Therefore, the feasibility of detecting those characteristics has to be studied in further research. The first indication for an ongoing attack is the type of overlays that are drawn. The overlays from this implementation are either of `TYPE_TOAST` or `TYPE_SYSTEM_ALERT` overlays. Furthermore, as Yan et al. discovered, many malware samples use `TYPE_SYSTEM_ERROR` overlays as well [13]. The type of an overlay, therefore, indicates an ongoing attack.

Moreover, the appearance of the overlays could be evaluated. If the overlays are transparent, it is very likely that they are used in a malicious way as they are hidden from the user. However, this characteristic is challenging to detect as it mostly relies on the alpha value of the used color [13]. This will likely produce false positives as there are many legitimate uses of transparent or semi-transparent overlays, such as the *Twilight* app that filters out the blue color in order to preserve the natural sleep cycle [88]. Additionally, we showed that programmatically visible overlays are practically invisible to the user (Section 6.3.1) by setting the width to zero. These overlays can therefore have any color and bypass detection techniques that rely on transparency detection. The dimensions of an overlay, such as its width and its height, therefore have to be considered as well.

The third characteristic is the number of drawn overlays. To reduce possible permutations, an adversary has to utilize a high number of overlays as described in Section 5.2. In order to make the attack feasible, an adversary needs to draw at least one overlay for each key on the keyboard. This results in at least 26 overlays as estimated in Section 5.2. A legitimate application will most likely not use such high amounts of overlays and therefore restrict to just a few.

The number of overlays can be correlated with their position in order to strengthen the suspicion of an ongoing attack. Most keyboards are drawn in the bottom area of the screen. The overlays are therefore mostly placed in the bottom region of the screen where the keyboard is placed. If many overlays are drawn at the bottom area of the screen, an ongoing attack is very likely. Additionally, all overlays form a grid and, as a result, inherit distinct locations on the screen without overlapping each other.

The last characteristic is the usage of permissions. In order to maximize the effectiveness of the attack, an adversary has to guarantee that the application is executed in the background. Thus, it should contain the capability of restarting in case it was forcibly closed or the device restarted. This process needs additional permissions, such as the `RECEIVE_BOOT_COMPLETED` permission. Therefore, the usage of certain permissions may also be an indicator for the attack.

10. Conclusion and Future Work

Mobile phones are playing an extremely important role in our everyday life. Most of these devices operate under the Android Operating System and are used for highly sensitive tasks such as Online Banking. Ensuring their security is therefore of great importance. Overlay-based attacks pose one of the biggest threats as they are highly versatile and challenging to detect. They aim at exploiting legitimate features of the User Interface to steal sensitive information.

In this thesis, we closely examined an important type of attack against Android smartphones that exploit weaknesses in the user interface. In particular, we studied, analyzed, and implemented the “Invisible Grid” attack from the “Cloak and Dagger” paper [1, 2] that enables an attacker to steal sensitive keyboard input like passwords. The attack takes advantage of a vulnerability that was subsequently patched on almost all Android versions. Nevertheless, it still affects a significant number of smartphone users with older devices as they are no longer entitled to security fixes. These devices are therefore unpatched and vulnerable to exploitation.

The implemented attack serves as a malicious to facilitate the development of future detection methods, for instance, by using artificial intelligence methods. The main target is to produce a malicious execution trace that will be utilized in a machine learning project that aims at detecting overlay-based malware. Results acquired by this thesis can also help to better understand the individual steps of an attack as well as the characteristics that could be used to create specific features helpful for attack detection.

We presented our concept and elaborated on the in-depth details of the implementation. We developed additional approaches and solved technical challenges to make the end-to-end attack feasible. Three methods to detect the presence of a keyboard were presented and compared. Additionally, we developed a module capable of dynamically retrieving layout files from the Android Open Source Project keyboard, making the attack straightforward and applicable to more than seventy languages. Furthermore, we developed a static approach that utilizes predefined layout files in order to attack almost any keyboard. In order to perfectly place the attack on a device, another module aiming at merging all information from previous modules was developed as well. This module is capable of detecting the areas in which clicks occur. The implementation itself is able to mimic the behavior of a keyboard and perfectly utilize the attack to monitor sensitive keyboard input.

We presented our novel defense technique `OverlayShifter` which is capable of preventing the attack while preserving usability and overlay functionality. We have elaborated that the defense technique shifts all adversarial overlays below the keyboard and therefore prevents a successful attack. Furthermore, we have explained implementational details and discussed future improvements.

An evaluation has shown that the implementation is applicable to a wide range of Android versions. Certain characteristics that facilitate the detection of an attack were interpreted as well. Furthermore, we have discussed countermeasures, apart from our developed novel defense technique, such as in-app keyboards.

Future Work We briefly discuss a follow-up attack that could be implemented by using our research results. The *Window Method* that was presented in Section 6.3.1 could be used to infer the exact moment when users begin to type into sensitive input fields, for instance, a login screen. An adversary may utilize it to launch high precision input hijacking attacks by spawning an identical login overlay in the right moment without the user's knowledge. The feasibility of this follow-up attack should be investigated in future research.

The developed novel defense method `OverlayShifter` was only presented in a very basic version. Future work should focus on improvements such as performance optimization.

The attack does not work on smartphones running a more recent Android version than 7.1. A significant share of users, however, is still affected by such attacks as they are utilizing older devices. Future work, therefore, has to focus on mitigating the consequences of an attack for affected users, e.g. by using alternative user interface elements that do not limit capabilities but preserve confidentiality and integrity as well. Moreover, advanced detection techniques that do not rely on OS modification should be studied to provide safety for devices that are no longer entitled to security fixes.

List of Figures

List of Tables

Listings

Acronyms

AOSP Android Open Source Project

XML eXtensible Markup Language

UI User Interface

UID Unique IDentifier

ID IDentifier

API Application Programming Interface

CVE Common Vulnerabilities and Exposures

PIN Personal Identification Number

OS Operating System

dp Density-independent Pixels

VM Virtual Machine

Bibliography

[1] Y. Fratantonio, C. Qian, S. Chung, and W. Lee, "Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop," in *2017 IEEE Symposium on Security and Privacy (Oakland)*, (San Jose, CA), May 2017.

[2] Y. Fratantonio, C. Qian, S. Chung, and W. Lee, "Cloak & Dagger." `https://cloak-and-dagger.org/#Attacks`, 05 2017. Accessed 2020-09-20.

[3] StatCounter GlobalStats. `https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide/#monthly-202004-202005-bar`, 05 2020. Accessed 2020-09-20.

[4] Check Point Software Technologies Ltd. `https://blog.checkpoint.com/2016/07/01/from-hummingbad-to-worse-new-in-depth-details-and-analysis-of-the-hummingbad-andriod-malware-campaign/`, 02 2016. Accessed 2020-09-20.

[5] D. Maslennikov, "ZeuS-in-the-Mobile – Facts and Theories." `https://securelist.com/zeus-in-the-mobile-facts-and-theories/36424/`, 08 2011. Accessed 2020-09-20.

[6] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege Escalation Attacks on Android," in *Information Security*, pp. 346–360, Springer Berlin Heidelberg, 2011.

[7] O. S. Adebayo and N. A. Aziz, "Techniques for analysing Android malware," in *2014 International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, 2014.

[8] X. Li, J. Liu, Y. Huo, R. Zhang, and Y. Yao, "An Android malware detection method based on AndroidManifest file," in *2016 International Conference on Cloud Computing and Intelligence Systems (CCIS)*, 2016.

[9] M. Karami, M. Elsabagh, P. Najafiborazjani, and A. Stavrou, "Behavioral Analysis of Android Applications Using Automated Instrumentation," in *2013 IEEE International Conference on Software Security and Reliability Companion*, 2013.

[10] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson, "Clickjacking: Attacks and Defenses," in *2012 USENIX Security Symposium (USENIX Security 12)*, pp. 413–428, 2012.

[11] E. Fernandes, Q. A. Chen, J. Paupore, G. Essl, J. Halderman, Z. Mao, and A. Prakash, "Android UI Deception Revisited: Attacks and Defenses," in *Financial Cryptography and Data Security*, pp. 41–59, Springer Berlin Heidelberg, 2017.

[12] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, "What the App is That? Deception and Countermeasures in the Android User Interface," in *2015 IEEE Symposium on Security and Privacy*, 2015.

[13] Y. Yan, Z. Li, Q. A. Chen, C. Wilson, T. Xu, E. Zhai, Y. Li, and Y. Liu, "Understanding and Detecting Overlay-Based Android Malware at Market Scales," in *2019 Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[14] Google, "Application Sandbox." `https://source.android.com/security/app-sandbox`, 06 2020. Accessed 2020-09-20.

[15] Google, "Permissions on android." `https://developer.android.com/guide/topics/permissions/overview`, 11 2020. Accessed 2020-11-21.

[16] Google, "Activity." `https://developer.android.com/reference/android/app/Activity`, 08 2020. Accessed 2020-09-20.

[17] Google, "Window." `https://developer.android.com/reference/android/view/Window`, 06 2020. Accessed 2020-09-20.

[18] Google, "View." `https://developer.android.com/reference/android/view/View`, 08 2020. Accessed 2020-09-20.

[19] Google, "WindowManager.LayoutParams." `https://developer.android.com/reference/android/view/WindowManager.LayoutParams`, 08 2020. Accessed 2020-09-20.

[20] Wikipedia, "Z-order." `https://en.wikipedia.org/wiki/Z-order`, 08 2020. Accessed 2020-09-20.

[21] Google, "EditorInfo." `https://developer.android.com/reference/android/view/inputmethod/EditorInfo`, 06 2020. Accessed 2020-09-20.

[22] Google, "ExtractEditText." `https://developer.android.com/reference/android/inputmethodservice/ExtractEditText.html`, 06 2020. Accessed 2020-09-20.

[23] Google, "TextView." `https://developer.android.com/reference/android/widget/TextView#attr_android:inputType`, 08 2020. Accessed 2020-09-20.

[24] M. P., *Grundkurs Betriebssysteme: Architekturen, Betriebsmittelverwaltung, Synchronisation, Prozesskommunikation.* Vieweg, 2009.

[25] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into your app without actually seeing it: UI state inference and novel android attacks," in *2014 USENIX Security Symposium (USENIX Security 14)*, (San Diego, CA), pp. 1037–1052, USENIX Association, 2014.

[26] D. Akhawe, W. He, Z. Li, R. Moazzezi, and D. Song, "Clickjacking revisited: A perceptual view of UI security," in *2014 USENIX Workshop on Offensive Technologies (WOOT 14)*, (San Diego, CA), USENIX Association, 2014.

[27] Cong Zheng, Wenjun Hu, Xiao Zhang and Zhi Xu, "Android Toast Overlay Attack: "Cloak and Dagger" with No Permissions." `https://unit42.paloaltonetworks.com/unit42-android-toast-overlay-attack-cloak-and-dagger-with-no-permissions/`, 09 2017. Accessed 2020-11-29.

[28] E. Ulqinaku, L. Malisa, J. Stefa, A. Mei, and S. Capkun, "Using Hover to Compromise the Confidentiality of User Input on Android," in *2017 ACM Conference on Security and Privacy in Wireless and Mobile Networks*, p. 12–22, 07 2017.

[29] B. Hill, *Adaptive User Interface Randomization as an Anti-clickjacking Strategy.* 2012.

[30] A. AlJarrah and M. Shehab, "Maintaining User Interface Integrity on Android," in *2016 IEEE Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 449–458, 2016.

[31] A. Kalysch, D. Bove, and T. Müller, "How Android's UI Security is Undermined by Accessibility," in *2018 Reversing and Offensive-oriented Trends Symposium*, pp. 1–10, 11 2018.

[32] Google, "ViewManager." `https://developer.android.com/reference/android/view/ViewManager`, 08 2020. Accessed 2020-09-20.

[33] Google, "Manifest.permission." `https://developer.android.com/reference/android/Manifest.permission#SYSTEM_ALERT_WINDOW`, 06 2020. Accessed 2020-09-20.

[34] Google, "Provide advance notice to the Google Play App Review team." `https://support.google.com/googleplay/android-developer/contact/adv_note?hl=en`, 06 2020. Accessed 2020-09-20.

[35] Google, "Manifest.permission." `https://developer.android.com/reference/android/Manifest.permission`, 09 2020. Accessed 2020-09-20.

[36] Google, "WindowManager." `https://developer.android.com/reference/android/view/WindowManager`, 06 2020. Accessed 2020-09-20.

[37] Google, "The Android Open Source Project - MotionEvent.java." `https://android.googlesource.com/platform/frameworks/base/+/613f63b938145bb86cd64fe0752eaf5e99b5f628/core/java/android/view/MotionEvent.java#417`, 03 2009. Accessed 2020-09-20.

[38] C.-.-. Available from MITRE, "CVE-2017-0860.." `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0860`, 11 2017. Accessed 2020-09-20.

[39] S. Vishniakou, "Android Open Source Project." `https://android.googlesource.com/platform/frameworks/native/+/5508ca2c191f8fdf29d8898890a58bf1a3a225b3%5E%21#F0`, 08 2017. Accessed 2020-09-20.

[40] CBS Interactive. `https://www.zdnet.com/article/android-security-warning-one-billion-devices-no-longer-getting-updates/`, 03 2020. Accessed 2020-09-20.

[41] Google, "Pixel/Nexus Security Bulletin—November 2017." `https://source.android.com/security/bulletin/pixel/2017-11-01`, 11 2017. Accessed 2020-09-20.

[42] R. Whitwam, "Google Posts Android 5.1 Factory Images For The Nexus 4, Nexus 7 2013, And Nexus 7 2013 LTE." `https://www.androidpolice.com/2015/04/14/google-posts-android-5-1-factory-images-for-the-nexus-4-nexus-7-2013-and-nexus-7-2013-lte/`, 04 2015. Accessed 2020-09-20.

[43] Google, "Learn when you get Android updates on Nexus devices." `https://support.google.com/nexus/answer/4457705?hl=en`, 06 2020. Accessed 2020-09-20.

[44] Google, "Choreographer." `https://developer.android.com/reference/android/view/Choreographer`, 12 2019. Accessed 2020-09-20.

[45] Wikipedia, "Latin alphabet." `https://en.wikipedia.org/wiki/Latin_alphabet`, 08 2020. Accessed 2020-09-20.

[46] Google, “Specify the input method type.” https://developer.android.com/training/keyboard-input/style, 09 2019. Accessed 2020-09-20.

[47] Wikipedia, “Polling (computer science).” https://en.wikipedia.org/wiki/Polling_(computer_science), 09 2020. Accessed 2020-09-20.

[48] E. Gamma, R. Helm, and R. Johnson, *Design Patterns. Elements of Reusable Object-Oriented Software*, vol. 206. Addison-Wesley Longman Publishing Co., Inc., 01 1995.

[49] Google, “InputMethodService.” https://developer.android.com/reference/android/inputmethodservice/InputMethodService, 06 2020. Accessed 2020-09-20.

[50] Google, “InputMethod.” https://developer.android.com/reference/android/view/inputmethod/InputMethod, 12 2020. Accessed 2020-09-20.

[51] Google, “Settings.Secure.” https://developer.android.com/reference/android/provider/Settings.Secure, 08 2020. Accessed 2020-09-20.

[52] Google, “InputMethodSubtype.” https://developer.android.com/reference/android/view/inputmethod/InputMethodSubtype, 05 2020. Accessed 2020-09-20.

[53] Google, “InputMethodManager.” https://developer.android.com/reference/android/view/inputmethod/InputMethodManager, 06 2020. Accessed 2020-09-20.

[54] Google, “WindowManager.LayoutParams - TYPE_SYSTEM_ALERT.” https://developer.android.com/reference/android/view/WindowManager.LayoutParams#TYPE_SYSTEM_ALERT, 06 2020. Accessed 2020-09-20.

[55] Wikipedia, “Reflection (computer programming).” https://en.wikipedia.org/wiki/Reflection_(computer_programming), 09 2020. Accessed 2020-09-20.

[56] Google, “Android Open Source Project.” https://github.com/aosp-mirror/platform_frameworks_base/blob/master/core/java/android/view/inputmethod/InputMethodManager.java#L2946, 07 2020. Accessed 2020-09-20.

[57] Oracle, “Oracle Java Documentation - Class.” https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html, 2019. Accessed 2020-09-20.

[58] S. Brouwer, “Samples-keyboardheight.” https://github.com/siebeprojects/samples-keyboardheight, 06 2019. Accessed 2020-09-20.

[59] Google, “PopupWindow.” https://developer.android.com/reference/android/widget/PopupWindow, 05 2020. Accessed 2020-09-20.

[60] Google, “ViewTreeObserver.” https://developer.android.com/reference/android/view/ViewTreeObserver, 05 2020. Accessed 2020-09-20.

[61] Google, “Material Design - Accessibility.” https://material.io/design/usability/accessibility.html, 2020. Accessed 2020-09-20.

[62] Google, “Context.” https://developer.android.com/reference/android/content/Context, 09 2020. Accessed 2020-09-20.

[63] Google, “Intent.” https://developer.android.com/reference/android/content/Intent, 08 2020. Accessed 2020-09-20.

[64] Google, “Activity.” https://developer.android.com/reference/android/app/Activity, 08 2020. Accessed 2020-09-20.

[65] Google, “Keyboard.” https://developer.android.com/reference/android/inputmethodservice/Keyboard, 12 2019. Accessed 2020-09-20.

[66] Google, “App resources overview.” https://developer.android.com/guide/topics/resources/providing-resources, 09 2020. Accessed 2020-09-20.

[67] Google, “Set the application ID.” https://developer.android.com/studio/build/application-id, 08 2020. Accessed 2020-09-20.

[68] Wikipedia, “IETF language tag.” https://en.wikipedia.org/wiki/IETF_language_tag, 08 2020. Accessed 2020-09-20.

[69] Google, “App resources overview.” https://developer.android.com/guide/topics/resources/providing-resources, 09 2020. Accessed 2020-09-20.

[70] Google, “Android Open Source Project.” https://android.googlesource.com/platform/frameworks/base/+/master/core/res/res/values/dimens.xml, 2020. Accessed 2020-09-20.

[71] S. Kondik, “Lineage OS.” https://github.com/LineageOS/android_packages_inputmethods_LatinIME, 2020. Accessed 2020-09-20.

[72] “Gboard - the Google Keyboard.” https://play.google.com/store/apps/details?id=com.google.android.inputmethod.latin&hl=en, 2020. Accessed 2020-09-20.

[73] Google, “Android Open Source Project.” https://android.googlesource.com/platform/packages/inputmethods/LatinIME/+/refs/heads/nougat-release/java/res/xml/, 2016. Accessed 2020-09-20.

[74] Wikipedia, “Locale (computer software).” https://en.wikipedia.org/wiki/Locale_(computer_software), 06 2020. Accessed 2020-09-20.

[75] Google, “Package.” https://developer.android.com/reference/java/lang/Package, 12 2019. Accessed 2020-09-20.

[76] Google, “Resources.” https://developer.android.com/reference/android/content/res/Resources, 06 2020. Accessed 2020-09-20.

[77] Google, “Android Open Source Project.” https://android.googlesource.com/platform/packages/inputmethods/LatinIME/+/refs/heads/nougat-release/java/res/values/donottranslate.xml#81, 10 2014. Accessed 2020-09-20.

[78] D. Fischer, “Gboard: Umlaute auf der Google-Tastatur aktivieren.” https://www.smartdroid.de/gboard-umlaute-auf-der-google-tastatur-aktivieren/, 11 2018. Accessed 2020-09-20.

[79] Google, “The Android Open Source Project - KeyboardBuilder.java.” https://android.googlesource.com/platform/packages/inputmethods/LatinIME/+/refs/heads/nougat-mr2.3-release/java/src/com/android/inputmethod/keyboard/internal/KeyboardBuilder.java#237, 08 2012. Accessed 2020-09-20.

[80] B. Kaminski, M. Jakubczyk, and P. Szufel, “A framework for sensitivity analysis of decision trees,” in *2018 Central European Journal of Operations Research*, 05 2017.

[81] Google, "Android Open Source Project." `https://android.googlesource.com/platform/frameworks/base/+/refs/heads/nougat-release/core/res/AndroidManifest.xml#2159`, 2016. Accessed 2020-09-20.

[82] D. Florencio and C. Herley, "How to Login from an Internet Cafe Without Worrying about Keyloggers," 01 2006.

[83] Oracle, "Oracle Java Documentation - Intrinsic Locks and Synchronization." `https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html`, 2019. Accessed 2020-09-20.

[84] Google, "Services overview." `https://developer.android.com/guide/components/services`, 06 2020. Accessed 2020-09-20.

[85] Y. Yukawa, "Android Open Source Project." `https://android.googlesource.com/platform/frameworks/base/+/refs/heads/marshmallow-release/core/java/android/view/ViewRootImpl.java#3701`, 2015. Accessed 2020-09-20.

[86] D. Cardinal, "How does swype really work?." `https://www.extremetech.com/extreme/97837-how-does-swype-really-work`, 09 2011. Accessed 2020-09-20.

[87] A. Sinkov, "Elementary Cryptanalysis: A Mathematical Approach," pp. 70–71, The Mathematical Association of America (Inc.), 1966.

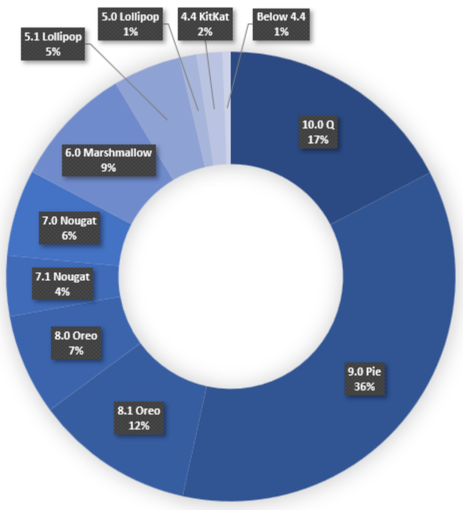
[88] P. Nálevka, "Twilight." `https://play.google.com/store/apps/details?id=com.urbandroid.lux&hl=en`. Accessed 2020-09-20.

[89] Y. Yukawa, "Android Open Source Project." `https://android.googlesource.com/platform/packages/inputmethods/LatinIME/+/refs/heads/master/java/res/xml/method.xml`, 2016. Accessed 2020-09-20.

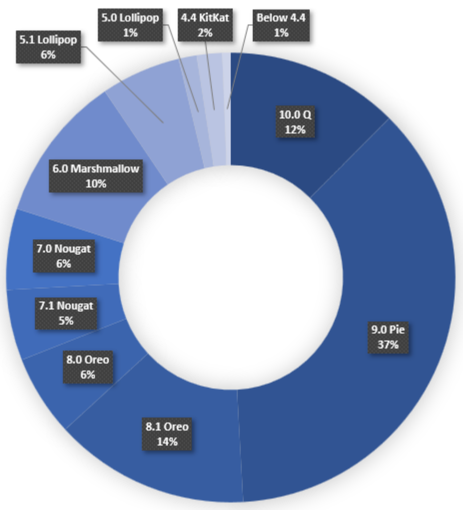
[90] StatCounter GlobalStats. `https://gs.statcounter.com/android-version-market-share/mobile-tablet/asia/#monthly-202004-202005-bar`, 05 2020. Accessed 2020-09-20.

Appendix

A. Appendix Market Share



(a) Android version market share in May 2020 (Worldwide) [3]



(b) Android version market share in May 2020 (Asia) [90]

Figure A.1.: Android Market Share

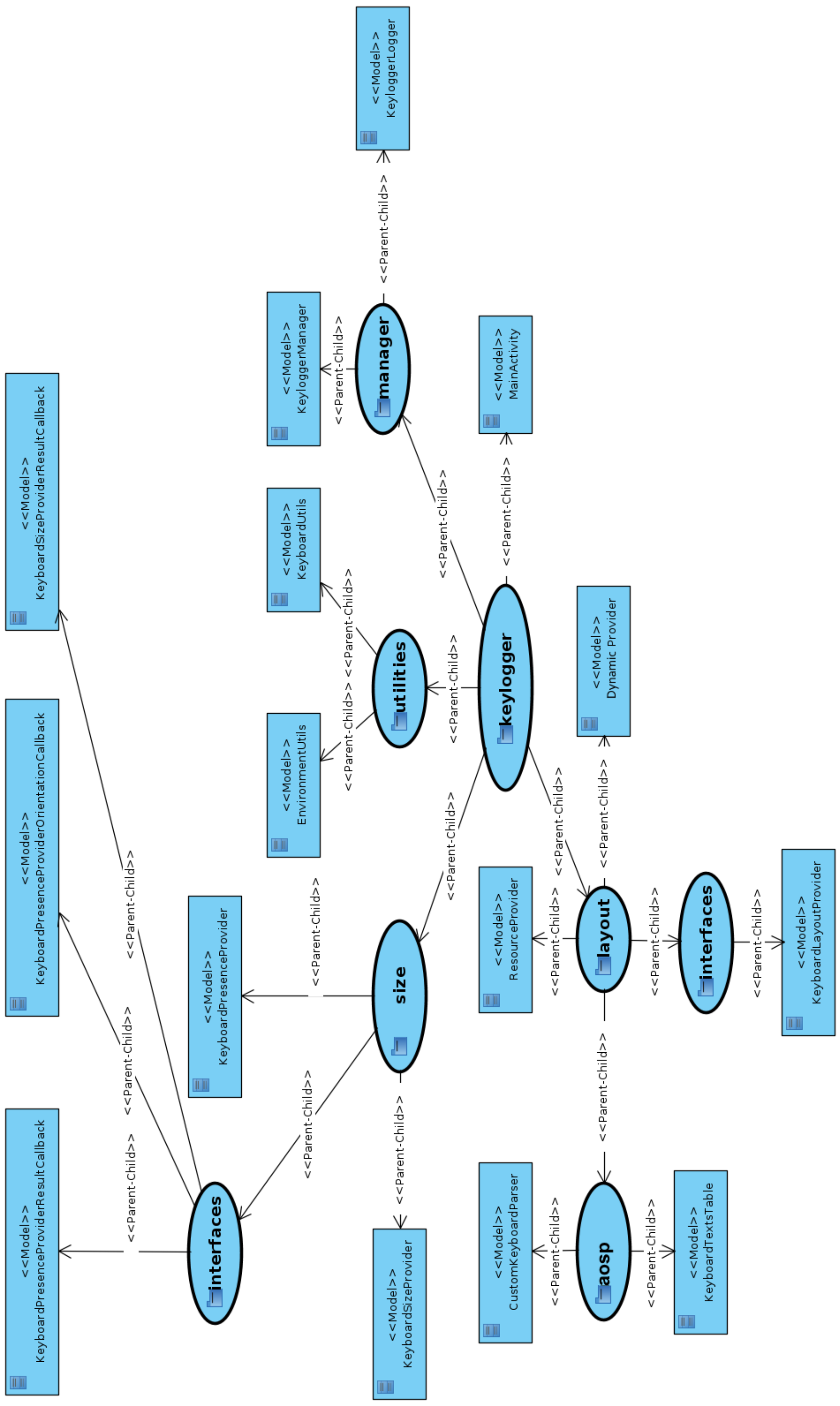
B. Appendix Implementation

Listing 10.1: Specially Crafted Overlay used by Window Method

```
View detectKeyBoardOpenView = View.inflate(ctx,
    ↪ R.layout.top_overlay_style, null);
LayoutParams layoutParams_detectKeyBoardOpenView = new
    ↪ WindowManager.LayoutParams(0, metrics.heightPixels,
    ↪ LayoutParams.TYPE_TOAST,
    ↪ WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE,
    ↪ PixelFormat.TRANSLUCENT);
//Add the View with the window manager
manager.addView(detectKeyBoardOpenView,
    ↪ layoutParams_detectKeyBoardOpenView);
//Add a popup window with a linear Layout to the view with the flag
    ↪ SOFT_INPUT_ADJUST_RESIZE to detect resize changes
PopupWindow popup = new PopupWindow();
LinearLayout detectSizeChangeLayout = new LinearLayout(ctx);
detectSizeChangeLayout.setLayoutParams(new LinearLayout.LayoutParams(
ViewGroup.LayoutParams.MATCH_PARENT,
    ↪ ViewGroup.LayoutParams.MATCH_PARENT));
//Add the linear layout to the popup View
popup.setContentView(detectSizeChangeLayout);
//Set SOFT_INPUT_ADJUST_RESIZE flag
popup.setSoftInputMode(
LayoutParams.SOFT_INPUT_ADJUST_RESIZE |
LayoutParams.SOFT_INPUT_STATE_ALWAYS_VISIBLE);
popup.setInputMethodMode(PopupWindow.INPUT_METHOD_NEEDED);
popup.setHeight(ViewGroup.LayoutParams.MATCH_PARENT);
//Add the popup to the detectKeyBoardOpenView overlay
detectKeyBoardOpenView.post(() ->
    ↪ popup.showAtLocation(detectKeyBoardOpenView, Gravity.NO_GRAVITY,
    ↪ 0, 0));
```

AOSP Label	**Implementation Label**
currencyKeyStyle	$
moreCurrency1KeyStyle	£
moreCurrency2KeyStyle	¢
moreCurrency3KeyStyle	€
moreCurrency4KeyStyle	¥
lessKeyStyle	<
greaterKeyStyle	>
spaceKeyStyle	[SPACE]
shiftKeyStyle	[SHIFT]
deleteKeyStyle	[DEL]
enterKeyStyle	[ENTER]

Table B.1.: Key label replacement of AOSP keyboard



<<Model>> KeyloggerLogger
<<Parent-Child>>
manager
<<Model>> KeyloggerManager
<<Model>> MainActivity
<<Model>> KeyboardUtils
utilities
<<Model>> EnvironmentUtils
keylogger
<<Model>> Dynamic Provider
size
<<Model>> KeyboardPresenceProvider
layout
<<Model>> ResourceProvider
interfaces
<<Model>> KeyboardLayoutProvider
<<Model>> KeyboardSizeProviderResultCallback
<<Model>> KeyboardPresenceProviderOrientationCallback
<<Model>> KeyboardPresenceProviderResultCallback
interfaces
aosp
<<Model>> CustomKeyboardParser
<<Model>> KeyboardTextsTable
<<Model>> KeyboardSizeProvider

Figure B.2.: The Java Package diagram of the app

Listing 10.2: Selected lines from Google Gboard englisch attack layout

```
<Row android:keyWidth="10%p" android:keyHeight="9%p"
    ↪ android:keyboardMode="
        @integer/keyboard_normal_portrait">
        <Key
                android:keyLabel="[SHIFT]"
                android:keyWidth="15%p" />
        <Key
                android:keyLabel="z" />
        <Key
                android:keyLabel="x" />
        <Key
                android:keyLabel="c" />
        <Key
                android:keyLabel="v" />
        <Key
                android:keyLabel="b" />
        <Key
                android:keyLabel="n" />
        <Key
                android:keyLabel="m" />
        <Key
                android:keyLabel="[DEL]"
                android:keyWidth="15%p" />
</Row>
<Row android:keyWidth="10%p" android:keyHeight="8%p"
    ↪ android:keyboardMode="
        @integer/keyboard_normal_portrait">
        <Key
                android:keyLabel="[Switch]"
                android:popupKeyboard="
                @integer/keyboard_symbol_portrait_pageOne"
                android:keyWidth="15%p" />
        <Key
                android:keyLabel="," />
        <Key
                android:keyLabel="[Language]" />
        <Key
                android:keyLabel="[Space]"
                android:keyWidth="40%p" />
        <Key
                android:keyLabel="."/>
        <Key
                android:keyLabel="[Enter]"
                android:keyWidth="15%p" />
</Row>
```

- Afrikaans
- Arabic
- Armenian (Armenia) Phonetic
- Azerbaijani (Azerbaijan)
- Belarusian (Belarus)
- Bulgarian
- Bengali (Bangladesh)
- Bengali (India)
- Catalan
- Croatian
- Czech
- Danish
- Dutch
- Dutch (Belgium)
- English (India)
- English (United States)
- English (Great Britain)
- Esperanto
- Spanish
- Spanish (United States)
- Spanish (Latin America)
- Estonian (Estonia)
- Basque (Spain)
- Persian
- Finnish
- French
- French (Canada)
- French (Switzerland)
- German
- Georgian (Georgia)
- German (Switzerland)
- Greek
- Galician (Spain)
- Hebrew
- Hindi
- Hinglish
- Hungarian
- Indonesian
- Icelandic
- Italian
- Italian (Switzerland)
- Kazakh
- Khmer (Cambodia)
- Kannada (India)
- Kyrgyz
- Lao (Laos)
- Lithuanian
- Latvian
- Macedonian
- Malayalam (India)
- Mongolian (Mongolia)
- Marathi (India)
- Malay (Malaysia)
- Norwegian Bokmål
- Nepali (Nepal) Romanized
- Nepali (Nepal) Traditional
- Polish
- Portuguese (Brazil)
- Portuguese (Portugal)
- Romanian
- Russian
- Sinhala (Sri Lanka)
- Slovak
- Slovenian
- Serbian
- Serbian (Latin)
- Swedish
- Swahili
- Tamil (India)
- Tamil (Sri Lanka)
- Tamil (Singapore)
- Telugu (India)
- Thai
- Tagalog
- Turkish
- Ukrainian
- Uzbek (Uzbekistan)
- Vietnamese
- Zulu

Table B.2.: Supported languages / layouts by the Android Open Source Project keyboard [89]

C. Appendix OverlayShifter

Listing 10.3: Core Logic of OverlayShifter

```java
private Context ctx=this; // The current Context
private View[]overlay=new View[4];
private WindowManager manager;
private WindowManager.LayoutParams layoutParams=new
    ↪ WindowManager.LayoutParams(0,0,
        WindowManager.LayoutParams.TYPE_SYSTEM_ERROR,FLAG_NOT_FOCUSABLE|
        FLAG_NOT_TOUCHABLE,PixelFormat.TRANSLUCENT);

private void shiftAllOverlays(int overlayID){
        View previousOverlay;
// Flag the second previous overlay for removal
        int previousOverlayIndex=overlayID-2;
        if(previousOverlayIndex< 0)previousOverlayIndex+=3;
        previousOverlay=overlay[previousOverlayIndex];
// Create new overlay
        View newOverlay=new View(ctx);
// Add the new overlay
        ((Activity)ctx).runOnUiThread(()->{
        manager.addView(newOverlay,layoutParams);
        overlay[overlayID]=newOverlay;});
// Create popup
        PopupWindow popup=new PopupWindow();
        TextView emptyTextView=new TextView(ctx);
// Add, for instance, a TextView as content (will be invisible anyways)
        popup.setContentView(emptyTextView);
// Set popup flags
        popup.setSoftInputMode(
        WindowManager.LayoutParams.SOFT_INPUT_ADJUST_RESIZE|
        WindowManager.LayoutParams.SOFT_INPUT_STATE_ALWAYS_VISIBLE);
        popup.setInputMethodMode(PopupWindow.INPUT_METHOD_NEEDED);
// Add the popup to the overlay
        newOverlay.post(()->popup.showAtLocation(
                        newOverlay,Gravity.NO_GRAVITY,0,0));
// Remove the overlay which was flagged for removal
        if(previousOverlay!=null){
        ((Activity)ctx).runOnUiThread(()->
                        manager.removeView(previousOverlay));
        }
}
```

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Würzburg, 14 December 2020

. . . .

(Jasper Stang)