TapTrap: Animation-Driven Tapjacking on Android

Philipp Beer
TU Wien

Marco Squarcina
TU Wien

Sebastian Roth
University of Bayreuth

Martina Lindorfer
TU Wien

Abstract

Users interact with mobile devices under the assumption that the graphical user interface (GUI) accurately reflects their actions, a trust fundamental to the user experience. In this work, we present *TapTrap*, a novel attack that enables *zero-permission* apps to exploit UI animations to undermine this trust relationship. TapTrap can be used by a malicious app to stealthily bypass Android's permission system and gain access to sensitive data or execute destructive actions, such as wiping the device without user approval. Its impact extends beyond the Android ecosystem, enabling tapjacking and Web clickjacking. TapTrap is able to bypass existing tapjacking defenses, as those are targeted toward overlays. Our novel approach, instead, abuses activity transition animations and is effective even on Android 15. We analyzed 99,705 apps from the Play Store to assess whether TapTrap is actively exploited in the wild. Our analysis found no evidence of such exploitation. Additionally, we conducted a large-scale study on these apps and discovered that 76.3% of apps are vulnerable to TapTrap. Finally, we evaluated the real-world feasibility of TapTrap through a user study with 20 participants, showing that all of them failed to notice at least one attack variant. Our findings have resulted in two assigned CVEs.

1 Introduction

Mobile devices are now an integral part of everyday life. They provide users with news, entertainment, communication, healthcare, banking, and other sensitive services. Fundamental to this user experience is the device's graphical user interface (GUI). As the GUI is the user's primary means of interaction with the device, it carries high trust. Users interact with the GUI under the assumption that it accurately reflects the underlying state of the device and that GUI elements, such as buttons, correspond to the actions they intend to perform.

Threat actors have long sought to undermine this trust relationship [2, 31, 36, 71, 74]. By creating malicious overlays on top of the GUI, attackers can trick users into performing unintended actions, such as authorizing financial transactions or granting sensitive permissions. This type of attack is commonly known as *tapjacking*. Several strategies have been added to Android over the years to counter this threat. These include restrictions on the `SYSTEM_ALERT_WINDOW` permission, mechanisms to automatically dismiss overlays during sensitive interactions like permission prompts, and other defenses introduced by default in Android 12. These mitigations, however, only target known tapjacking techniques using overlays.

In this work, we present *TapTrap*, a novel tapjacking attack on the Android system. Compared to previous tapjacking attacks that rely on malicious overlays, TapTrap leverages a previously unexplored and fundamentally different mechanism: activity transition animations. By exploiting these animations, a core feature of the Android UI experience, a *zero-permission app* can spawn a benign transparent activity on top of a malicious one, therefore bypassing existing tapjacking defenses at the system and app levels. This enables a malicious app to stealthily bypass Android's permission system and gain unauthorized access to sensitive user data, such as location, camera, and notification content. Beyond data access, we demonstrate that TapTrap can escalate its impact to compromise the entire device by triggering a factory reset without user approval. Additionally, we show that the attack also threatens the user's security and privacy on the Web by enabling clickjacking, highlighting its broader applicability.

We investigate whether this vulnerability is actively exploited in the wild based on 99,705 apps that we downloaded from the Google Play Store and find no evidence of TapTrap exploitation, suggesting that this attack is a previously unexplored threat vector. Additionally, to assess the impact of TapTrap, we perform further analyses on these apps, revealing that 76.3% of the analyzed apps are vulnerable to the attack.

Finally, to evaluate the real-world feasibility of TapTrap, we conduct a carefully designed ethical study with 20 participants. Using a custom game app to simulate the attack, we find that all participants failed to notice at least one attack variant, underscoring the practical impact of TapTrap and the urgent need for effective defenses against this novel attack vector.

In summary, our work makes the following contributions:

- We present TapTrap, a novel attack on the Android system that can be used to bypass the Android permission system, perform Web clickjacking attacks, and can escalate to a full device erasure (Sec. 3).
- We thoroughly review existing literature on tapjacking attacks and classify previous work based on the exploited mechanisms and the capabilities of the attacks. TapTrap is the first to exploit activity transition animations and the only tapjacking attack that affects the current Android version. We also evaluate existing mitigations and propose effective solutions against TapTrap (Sec. 4).
- We analyze 99,705 apps and find no evidence of TapTrap exploitation in the wild. Using this analysis, we identify an issue in the Android platform that allows animations to run longer than expected, significantly extending the attack window of TapTrap (Sec. 5).
- We perform a large-scale evaluation of these apps, revealing that 76,035 are vulnerable to TapTrap (Sec. 6).
- We evaluate the real-world feasibility of TapTrap through a user study. All of the 20 participants failed to notice at least one attack variant, even after being informed of the possibility of an attack (Sec. 7).

Ethics and Disclosure. We responsibly disclosed TapTrap to the Android Security Team and the affected browser vendors. Google and all browser vendors acknowledged the issue. While browser vendors have since fixed the issue, even the current Android 15 remains vulnerable, as we discuss in more detail in the ethics section at the end of the paper.

2 Background

This section introduces the Android GUI elements relevant to our work, explains their interactions, and briefly summarizes permission model employed by the operating system.

2.1 Android GUI Elements and Interaction

Android apps are designed to enable interactions both within a single app and between different apps, allowing one app to invoke the functionalities of another. Below, we outline the core concepts of these interactions.

Activities. *Activities* [11] are one of the main building blocks of Android apps and provide the visual components for user interaction. In a typical Android app, each activity represents one screen of an app. An app can consist of multiple activities, where one activity can start another one using *intents*.

Intents. Intents [10] are Android's inter-component communication mechanism. They can be used to start other activities or components. Intents can either be explicit or implicit. In explicit intents, the target component is explicitly defined by its class name. In contrast, in implicit intents, the system determines the target component based on parameters such as the intent's action (e.g., opening a web browser for a URL).

Back Stack. In a typical setting, only one activity is visible to the user at a time and remains in the foreground. The system keeps track of activities in a so-called back stack [15]. When an activity is started from another activity, it is pushed onto the back stack, and the previous activity is stopped. When navigating back, the activity is popped from the back stack, and the previous activity is resumed. The Android system maintains a separate back stack for each *task*.

Task. A task represents a collection of activities when navigating through apps. Conceptually, it represents one item in the recent apps list. By default, when an activity is started from another activity, it is launched in the same task as the activity that started it. However, activities can also define that they can only be launched in a new task.

2.2 Android Permission Model

Each Android app operates within its own kernel-level sandbox, running as a separate process [21]. This ensures that apps are isolated from each other and cannot access other apps' data or system resources without proper authorization. Android employs a permission system to regulate access to sensitive resources based on three categories: install-time permissions, runtime permissions, and special permissions [13].

Install-time Permissions. Install-time permissions are automatically granted when the user installs an app. These permissions are typically used for non-sensitive operations that do not require explicit user approval.

Runtime Permissions. Android requires apps to request runtime permissions for potentially dangerous actions, such as accessing the camera or retrieving the device's location. These permissions are presented to the user via a system dialog when the application attempts to perform the relevant action, allowing the user to approve or deny the request at runtime.

Special Permissions. Special permissions differ from runtime permissions and must be explicitly granted by the user in the device settings. These permissions often involve significant security implications, and the user is explicitly warned about their potential impact before confirming the action. Examples of special permissions include the `SYSTEM_ALERT_WINDOW` permission that allows apps to draw over other apps and that has been frequently abused for tapjacking, the `BIND_ACCESSIBILITY_SERVICE` permission that enables apps to perform actions on behalf of the user for accessibility purposes, and the `BIND_DEVICE_ADMIN` permission that can be used to, e.g., remotely wipe the device.

3 TapTrap

In this section, we introduce TapTrap, an animation-based tapjacking attack on Android. Due to its unique mechanism, TapTrap bypasses existing tapjacking mitigations implemented by the Android system and apps. We outline the threat model, describe the attack mechanism, and present scenarios that demonstrate its practical impact, ranging from bypassing the Android permission model to enabling Web clickjacking.

3.1 Threat Model

We assume that a malicious app is installed on the user's device, for example, via an app market, as is typical in tapjacking scenarios [36, 71]. The app does not require any permissions, making it appear harmless and non-intrusive to the user. Notably, we do *not* assume the `SYSTEM_ALERT_WINDOW` permission, which prior work [31, 36] often relied on. However, we assume that animations are enabled on the device. By default, animations are active unless explicitly disabled through developer options or accessibility settings.

3.2 Attack Mechanism

When one activity starts another using the `startActivity` method or a related method, Android uses an animation to transition between the activities. The type of animation depends on various factors, such as whether the activities belong to the same or different tasks or whether the app's theme specifies certain animations. Transitions across different tasks always use a system-defined slide animation. If a transition occurs between activities in the same task, developers can specify custom animations using the `overridePendingTransition` method or the `makeCustomAnimation` method.

While this customization provides developers with fine-grained control over the app's appearance, we found that it can be exploited by malicious apps to lure users into performing sensitive actions without their knowledge. Our attack called TapTrap achieves this goal by creating a mismatch between what is displayed on the screen and the app's actual state.

The attack mechanism is illustrated in Fig. 1. Activity *A* of the malicious app starts activity *B* of a benign app containing a sensitive UI element, such as a button to confirm a transaction. *A* uses a carefully crafted animation to make activity *B* fully transparent by setting its starting and ending `alpha` values near 0 (e.g., 0.01). Additional animation effects, such as scaling and transformation, can be applied to zoom into activity *B*, making the sensitive element occupy the whole screen. Although *A* appears visible to the user, *B* is on top of the stack and handles touch events. The malicious activity *A* can place a UI element anywhere on the screen to lure the user into interacting with it, unknowingly triggering actions on *B*. Before the animation ends and *B* becomes visible, *A* relaunches itself and puts *B* into the background. Even though

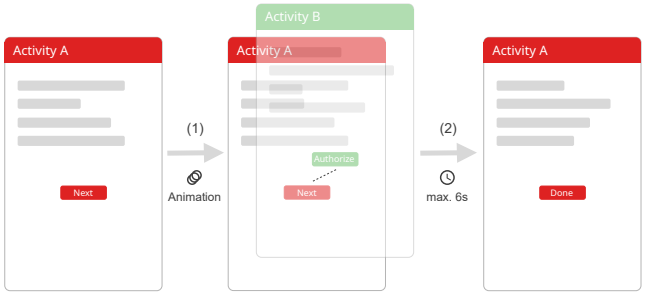


Figure 1: Overview of TapTrap. (1) Activity *A* of a malicious app starts activity *B* of a benign app using a low-opacity animation. *B* is on top and reactive to touches, but *A* remains visible, luring the user into unknowingly interacting with *B*. (2) *A* is relaunched and hides *B* before the animation ends.

Android introduced restrictions on background launches for apps targeting Android 14 and above, activity launches used for TapTrap are not blocked[1] [20].

Attack Window. TapTrap's attack window is limited to 3 seconds, which corresponds to the maximum duration for activity transition animations. However, as we discuss in Sec. 5, we discovered a flaw in the implementation of this restriction, allowing animations to run for up to 6 seconds, doubling the attack window. Importantly, TapTrap remains effective even without this flaw. If the attack does not succeed within the attack window, the app can restart it.

Attack Implementation. We provide a minimal implementation of TapTrap in Appendix A. A full proof-of-concept, including the restart strategy, is included in the artifacts.

3.3 Attacking the Android System

In this section, we report on selected case studies to demonstrate the impact of TapTrap, ranging from bypassing Android's runtime permission model to full device erasure.

3.3.1 Circumventing Runtime Permissions

As discussed in Sec. 2.2, Android's permission model restricts access to sensitive data and operations. To request a runtime permission, an activity must send an implicit intent with the `REQUEST_PERMISSIONS` action and specify the requested permissions using the `REQUEST_PERMISSIONS_NAME` intent extra. The framework provides APIs to abstract this process, making it seamless for developers. We found that the activity responsible for displaying the permission prompt is vulnerable to TapTrap as it can be opened in the same task as the malicious app and does not block custom animations. This vulnerability allows malicious apps to request arbitrary runtime permissions without the user's knowledge, such as access to the user's location, camera, contacts, or microphone.

[1] Tested on Android 15 using an app targeting the newest API Level 35.

Because the malicious activity that is visible is not on top of the stack during the animation, it does not receive touch events. Therefore, the activity cannot directly detect when the user taps the "allow" button. Instead, the activity can listen to the `onResume` lifecycle method. This method is triggered whenever an activity is brought to the foreground, such as when the prompt disappears because the user taps the "allow" button. This side channel can be used to detect the click and update the UI of the malicious activity accordingly, keeping the user unaware of the attack.

3.3.2 Notification Sniffing

The `BIND_NOTIFICATION_LISTENER_SERVICE` permission allows apps to read, reply to, and control notifications, making it a highly sensitive capability. Users must explicitly grant this permission in the device settings. A malicious app can programmatically open the corresponding settings screen by sending an intent with the `NOTIFICATION_LISTENER_DETAIL_SETTINGS` action and including the service containing the notification listener logic in the `NOTIFICATION_LISTENER_COMPONENT_NAME` extra.

Granting this permission requires two steps: the user must first toggle the permission and then accept a warning dialog describing the associated risks. Consequently, the attack requires two user taps to succeed. The second tap can be detected by monitoring the `onBind` method of the notification listener service, which is invoked when the permission is granted. However, the first tap cannot be directly detected. To overcome this limitation, the malicious app can predict when the user is likely to tap on the screen or rely on other side channels to detect clicks, such as accelerometer or gyroscope data, as has been explored in prior work [46, 64, 72].

Once the permission is granted, a malicious app can intercept sensitive notifications, which can then be exploited in phishing campaigns, ransomware attacks, or other malicious activities. Furthermore, this would also allow the attacker to steal two-factor authentication codes received via SMS or email, escalating the attack to account takeover.

3.3.3 Full Device Erasure

Device administrator apps [9] are a special type of app that can perform sensitive security-relevant operations on a user's device. Apps that want to become device administrators must request the `BIND_DEVICE_ADMIN` permission, implement a receiver that extends `DeviceAdminReceiver`, and declare its capabilities in the app's resources. Similar to the `BIND_NOTIFICATION_LISTENER_SERVICE` permission, users need to explicitly grant the permission in the device settings.

The device administrator API is still supported, and its capabilities remain significant as of Android 15 [44]. For example, an app with device administrator privileges can remotely lock or wipe the device. A malicious app can open the device admin permission prompt by sending an explicit intent targeting the `DeviceAdminAdd` activity and including its package name in the `DEVICE_ADMIN` extra. After tapping the "activate" button, the activity closes itself with a slide-out animation. Due to this animation, the device admin activity is briefly visible before the malicious app is fully in the foreground again.

To prevent this, the malicious app can lock the screen immediately after the permission is granted, simulating an app crash to hide its activity. Nevertheless, at that time, the malicious app has already gained full control.

3.4 Attacking the Browser

TapTrap's impact is not confined only to system apps but extends beyond. This section demonstrates how TapTrap can exploit another integral part of the system: the browser.

To open a website in a browser, an app can create an implicit intent with the `ACTION_VIEW` action and set the URL of the website as the intent's data, or start an explicit intent that specifies the browser's package name and activity.

With Chrome 45, Google introduced Custom Tabs [34], a feature that allows browsers to be embedded in apps. Custom Tabs provide the same browsing experience as a standalone browser while sharing cookies, permissions, and other state with the browser. This feature is designed to enhance the user experience, and developers can customize the Custom Tab to fit their app's theme. Furthermore, they allow developers to specify the opening and closing animations, enabling a seamless transition between the app and the browser. Because of this rather tight integration with the host app, Custom Tab-supported browsers open the Custom Tab activity in the same task as the launching app, making them vulnerable to TapTrap.

Two primary security risks arise: (1) permission bypass within the Web ecosystem and (2) Web clickjacking attacks.

In the following, we discuss these risks and evaluate the vulnerability of 10 popular mobile browsers to TapTrap. Our analysis includes the top nine browsers from StatCounter's ranking [66] that are available on the Play Store, plus Brave. Despite not being listed by StatCounter, we added Brave to the analysis, as it does not expose its user agent via HTTP headers (the mechanism StatCounter uses for ranking) but has over 100M downloads on the Play Store. Table 1 summarizes the findings. Notably, Opera and UC Browser are not vulnerable to TapTrap, as they do not support Custom Tabs and always open the browser in a separate task. Additionally, while Firefox and Brave are vulnerable to permission bypass, they require two user clicks to persist permissions. By default, these browsers grant permissions only temporarily unless the user explicitly selects the "remember" option.

Note that to minimize the loading times of websites, an app can use the `warmup` method to preload the browser and the `mayLaunchUrl` method to prefetch a website. This is useful for unstable connections when websites would otherwise require a loading time that exceeds TapTrap's attack window.

Browser	Permission	Clickjacking
Chrome (v130)	●	●
Samsung Internet (v26)	●	●
Opera (v85)	○	○
UC Browser (v13.7)	○	○
Firefox (v131)	◐	●
Edge (v130)	●	●
Yandex (v128)	●	●
Naver Whale (v3.5)	●	●
Coc Coc (v133)	●	●
Brave (v130)	◐	●

Table 1: Browser vulnerability among popular mobile browsers (● vulnerable, ○ not vulnerable, ◐ requires two user clicks to persist permission).

3.4.1 Web Permission Bypass

The Web platform has evolved to include a wide range of permissions, driven by the increasing demand for app-like capabilities on the Web. Interaction with sensitive APIs, such as the user's location, camera, or microphone, is regulated by the permission model of the browser. Additional features include access to the system clipboard, Bluetooth, payment handlers, and push notifications. Access to these critical APIs is programmatically requested by the website via the Permissions API [55], which displays a permission prompt to the user. Once granted, the website can access the requested API without further user interaction.

Attackers can stealthily obtain these permissions by loading a malicious website in a Custom Tab and leveraging TapTrap. While this may seem redundant given that TapTrap can be used to directly bypass Android-level permissions, exploiting the Web channel offers some advantages discussed below.

Unsuspicious App. The malicious app does not need to declare any Android permissions, creating the impression of a benign app and reducing user suspicion.

Transparency. Android 12 introduced the *Privacy Dashboard*, which allows users to see when apps access specific permissions in the device's settings [43]. Permissions granted to a website via TapTrap, however, are associated with the browser rather than the malicious app. Therefore, the Privacy Dashboard only lists the browser app but not the malicious app. Unless the user actively suspects a Web-based compromise and checks the browser's site-specific permissions, they are unlikely to detect the attack.

Persistence. Once granted, Web-based permissions persist even if the user uninstalls the malicious app. The website retains its privileges within the browser environment, providing attackers with a durable foothold in the user's Web ecosystem. For example, attackers could use the granted permissions to send phishing notifications long after the app is removed.

3.4.2 Clickjacking

Clickjacking, also known as *UI redressing*, is a well-known attack in the Web ecosystem. In a typical clickjacking scenario, an attacker embeds a victim website in a transparent iframe and tricks the user into clicking on elements of the embedded site while making it appear as though they are interacting with the attacker's page. This can result in unauthorized purchases, account takeovers, and data exfiltration [1, 32, 65].

Over the years, various techniques to mitigate clickjacking on the Web have been developed. Many of these defenses focus on framing protection mechanisms that prevent a site from being embedded in a malicious website, such as the `X-Frame-Options` header [56] and the `frame-ancestors` directive [54] of the `Content-Security-Policy` header.

SameSite cookies [57] provide another layer of protection. Introduced to primarily mitigate Cross-Site Request Forgery (CSRF) attacks, developers can set the `SameSite` attribute to `Lax` or `Strict` to restrict cookie transmission in cross-site contexts, such as when a page is embedded in an iframe. As a result, the embedded page does not receive such cookies and thus does not display authenticated content, preventing attackers from luring users into performing sensitive actions.

However, these Web-focused countermeasures assume that threats are confined to the Web environment. They do not account for interactions between browsers and the Android system, as navigations to websites loaded via Custom Tabs always attach `Lax` cookies. `Strict` SameSite cookies, instead, are not sent by default on Custom Tabs [26, 27]. Unfortunately, as Khodayari and Pellegrino [51] found, only 1,854 out of the top 500,000 websites (0.3%) employed `Strict` SameSite cookies as of 2021. More recent studies have confirmed that the `Strict` SameSite cookie adoption remains low (2%) [47], leaving the vast majority of sites vulnerable to TapTrap.

4 TapTrap's Unique Feature

To position TapTrap within the landscape of existing attacks, we begin by providing an overview of the attack vectors and capabilities of previous tapjacking attacks and compare them to TapTrap. Next, we examine how existing mitigations address known attacks and why they fail to protect against TapTrap. Finally, we propose new strategies to close the gaps in current mitigations and effectively defend against TapTrap.

4.1 Previous Tapjacking Attacks

Historically, tapjacking attacks have relied on malicious overlays drawn over a benign app. Depending on the specific type of attack, an attacker can achieve varying levels of control over the UI. In the following, we review existing literature on tapjacking attacks and classify the capabilities and attack vectors exploited by each attack. An overview of existing attacks and their classification is shown in Table 2.

We categorize **capabilities** into six different classes. Attackers can perform *app touchjacking*, i.e., luring users into interacting with a third-party app, or execute *permission bypass* attacks to trick users into granting sensitive permissions. Additionally, malicious apps can achieve *general touch-awareness*, allowing them to detect the timing and location of user touches. This capability can lead to *user input theft*, such as capturing passwords or other sensitive information using transparent overlays. Moreover, attackers may be able to *control UI elements*, i.e., arbitrarily position and size sensitive UI elements on the screen, increasing the flexibility of the attack.

Prior tapjacking **attack vectors** can be categorized into four classes: toast-based attacks, overlay window-based attacks, overlay activity-based attacks, and WebView-based attacks. We discuss each attack vector in detail below.

Toast-based Attacks. Toast-based attacks exploit a special type of overlay known as toasts. Toasts [17] are short-lived notifications that can be displayed on the screen for up to 3.5 seconds and do not require permissions to be used. Johnson [49] were among the first to demonstrate how this mechanism could be abused for tapjacking purposes in apps. Niemietz and Schwenk [59] further showed that toasts could be leveraged to perform Web clickjacking attacks, install other apps without user consent, and steal user input by monitoring touch events. More recent research [50, 70] has extended the scope of toast-based attacks and, among others, showed how toasts can be used to lure users into granting permission prompts in the browser. While all of these attacks have been mitigated at the latest with Android 12, Kar and Stakhanova [50] identified specific OEMs that were vulnerable even with Android 13.

WebView-based Attacks. Luo et al. [53] demonstrated a WebView-based attack where a malicious app embedded a WebView and overlaid it with arbitrary UI elements to trick users into unintended website interactions. However, its impact is limited as WebView's browsing context is sandboxed from the browser, and the attack is confined to WebViews.

Overlay Window-based Attacks. These attacks exploit the `SYSTEM_ALERT_WINDOW` permission, which must be granted through device settings and allows drawing windows over other apps. Ying et al. [74] showed that overlay windows can be used to hijack the system keyguard, i.e., the device lock pattern, or obscure permission prompts, tricking users into granting unintended runtime permissions. Although Android lacked runtime permissions at that time, the authors were able to circumvent custom ROMs developed by certain OEMs (e.g., Huawei). Finally, they show how to use such windows to infer numerical passwords by recording the time between taps. Fratantonio et al. [36] found that the `SYSTEM_ALERT_WINDOW` permission was automatically granted to apps downloaded from the Play Store. They showed how they can silently lure the user into granting accessibility permissions to the app and, in turn, take full control over the device. Cai et al. [31] used overlays to steal passwords, although a successful attack required users to type their passwords multiple times. Finally, Wang et al. [70] showed how rapidly creating and destroying overlay windows bypasses the notification visible when the permission is used, which was introduced in Android 8 [5], as we will describe later.

Overlay Activity-based Attacks. Other works, such as Alepis and Patsakis [2] and Wang et al. [71] have relied on using overlay activities to perform tapjacking attacks that do not require the `SYSTEM_ALERT_WINDOW` permission. While the first showed how they can partially overlay the device admin permission prompt and trick the user into granting it, the latter demonstrated how, due to missing mitigation in a specific settings screen, they can lure users into granting the "draw over apps" permission and circumvent physical input-based authorization, such as fingerprint sensors.

4.1.1 Key Differences to Previous Attacks

When comparing these previous attacks with TapTrap, we observe the following key differences:

Attack Feasibility. Compared to all previous attacks with similar capabilities that have been mitigated, TapTrap does not rely on overlays, the most common attack vector for tapjacking. This allows TapTrap, as we will see in the next section, to bypass existing mitigations tailored to preventing tapjacking attacks and to be effective even on Android 15. Like previous attacks, TapTrap is subject to runtime conditions (e.g., font size) that may influence UI element positions. However, most such parameters are accessible to an app, allowing it to adapt its layout dynamically. TapTrap's control over UI elements further increases tolerance to layout variation.

Control UI Elements. TapTrap is the only attack that has the capability to control the position of sensitive UI elements the user interacts with on the screen. This capability is further exacerbated by the ability to zoom into any part of the target activity, to the extent that the sensitive UI element, e.g., a payment button, can cover the entire screen. This is achieved by using translate and scale animations during the activity transition initiated by the malicious app.

Touch-Awareness. Compared to previous attacks, TapTrap does not have general touch awareness, i.e., it cannot detect taps on the screen. While, as we showed in the previous section, it can detect taps through additional side-channels, i.e., the `onResume` lifecycle method, it cannot detect taps on the screen in general. Therefore, TapTrap can also not be directly used to steal user input.

Persistence Across Activities. TapTrap cannot survive activity transitions. That means that if the user navigates to another activity within the benign app, the attack is interrupted. Therefore, TapTrap is not able to bypass the accessibility service permission and install other apps, as these actions require navigation across multiple activities.

Publication	Mechanism	Required Permission	App Touchjacking	Permission Bypass: Runtime	Device Admin	Display Over Apps	Accessibility	Notification Service	App Installation	Web	Clickjacking	Touch-awareness	Input Stealing	Control UI Elements	Affects Android 13+
Johnson [49] (2011)	Toast	–	●	–	○	–	○	○	○	○	○	○	○	○	○
Niemietz/Schwenk [59] (2012)	Toast	–	●	–	○	–	○	○	●	○	●	●	●	○	○
Luo et al. [53] (2012)	WebView	–	○	–	○	–	○	○	○	○	◐	◐	◐	◐	◐
Ying et al. [74] (2016)	Overlay Window	◆	●	◐	○	–	○	○	○	○	○	●	●	○	○
Fratantonio et al. [36] (2017)	Overlay Window	◈	●	●	●	–	●	●	●	○	●	●	●	○	○
Alepis/Patsakis [2] (2017)	Overlay Activity	–	●	○	●	○	○	○	●	○	○	●	●	–	○
Cai et al. [31] (2020)	Overlay Window	◆	○	○	○	○	○	○	○	○	○	–	●	–	○
Wang et al. [71] (2022)	Overlay Activity	–	●	○	○	●	○	○	○	○	○	○	○	○	○
Wang et al. [70] (2022)	Toast	–	●	○	○	○	○	○	○	○	○	🔗	🔗	○	○
Kar and Stakhanova [50] (2023)	Toast	–	○	○	○	○	○	○	○	●	○	◐	○	○	○
This work	Animation	–	●	●	●	●	○	●	○	●	●	○	○	●	●

Table 2: Comparison of TapTrap with previous attacks and what is covered in the respective publication (● yes, ○ no/not covered, ◐ restricted, – not applicable, ◆ `SYSTEM_ALERT_WINDOW` permission, ◈ `SYSTEM_ALERT_WINDOW` permission (but automatically granted by the Play Store), 🔗 when combined with overlay window and `SYSTEM_ALERT_WINDOW` permission).

4.2 Existing Mitigations

Android has introduced several mitigations to protect against tapjacking attacks. Certain protections are enabled by default, while others must be explicitly enabled by app developers. In the following, we provide an overview of existing tapjacking mitigations, how they mitigate previous attacks, and how TapTrap is able to bypass them. A timeline of the introduction for each mitigation in Android is outlined in Fig. 2.

4.2.1 Overlay Detection and Prevention

Android's primary defenses against tapjacking attacks focus on detecting and restricting overlays. In Android 2.3, the `FLAG_WINDOW_IS_OBSCURED` flag was introduced to detect when a touch event passes through an overlay, allowing security-sensitive applications to reject the touch O1.

This was later complemented by the `FLAG_WINDOW_IS_PARTIALLY_OBSCURED` [12] flag in Android 10, which protects against partial overlays, i.e., where only part of the UI element is overlaid O3. Additionally, the `setFilterTouchesWhenObscured` [18] method allows developers to filter out touch events passed through an overlay O2. At the latest with Android 7, security-critical setting screens were protected against tapjacking M1 [29]. Android 11 later introduced an overlay check that prevents overlays in the whole settings app M3, and Android 12 opened up this functionality to third-party apps through the `setHideOverlayWindows` method O4 [30]. From Android 6 onward, the `SYSTEM_ALERT_WINDOW` permission has to be explicitly granted in the device's settings M2. The permission was, however, automatically granted for apps downloaded from the Google Play Store [36]. For apps targeting Android 8 and upward, drawing over other apps shows a notification in the notification drawer S1 [5]. Finally, Android 12 adopted a system-wide tapjacking prevention mechanism that blocks touch interactions from overlays in most scenarios M4 [14].

All of these mechanisms fail to protect against TapTrap, as TapTrap does not rely on overlays.

4.2.2 Toast-specific Mitigations

Starting with Android 11, background apps are no longer allowed to display custom toasts, significantly reducing the risk of phishing attacks originating from background processes M5. Android 12 further enhanced this by completely blocking background toasts M7 and introducing mechanisms to prevent toast-burst attacks, where multiple toasts are displayed in succession M6. Additionally, toasts are now limited to two lines of text M8 [14, 16]. As TapTrap does not rely on toasts, these mitigations do not impact the attack.

4.2.3 User Awareness and Secure Interaction

Recent Android updates have introduced mechanisms to enhance user awareness and secure critical interactions. Android 9 introduced Protected Confirmation [6], a secure UI requiring explicit user confirmation for sensitive actions, such as financial transactions O5. Android 12 added the Privacy Indicator [23], a status bar feature that alerts users when apps access the camera or microphone S2. These strategies provide only limited protection against TapTrap. The Privacy

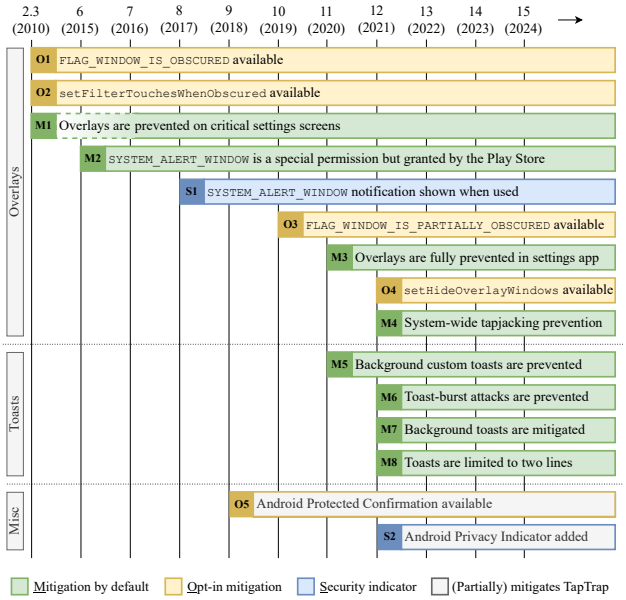


Figure 2: Android mitigations against tapjacking. The Android version refers to either the installed Android version or the app's target SDK.

Indicator covers only a small subset of permissions and, as we will show in Sec. 7, is not always effective. Furthermore, Protected Confirmation has been "deprecated due to [...] low adoption rate among app developers" [4].

4.3 Mitigations Against TapTrap

App developers can currently mitigate TapTrap by preventing sensitive activities from being launched with custom animations via the `overridePendingTransition` method or by deferring user input handling until the completion of enter animations using the `onEnterAnimationComplete` method. However, these measures apply only to third-party apps and do not address vulnerabilities in system components, such as permission prompts. Also, we believe that the responsibility for mitigating TapTrap should not rest on app developers but should instead be addressed at the system level.

A direct system-level solution is to prevent touch events during activity transition animations. While this approach would be effective, it risks negatively impacting the user experience as legitimate touches during animations are ignored. To address this, we propose a solution that accounts for the screen's opacity. Specifically, touch events should be blocked only when the animation's opacity falls below a defined threshold. Android 12's system-wide tapjacking defense defines a *maximum obscuring opacity* threshold of 0.8 that is used to determine whether overlays are allowed to pass touches through. Overlays below this are allowed to pass touches through, as the underlying UI elements are still visible [19]. Building on this concept, we propose applying an opacity threshold of 0.2 for activity transitions to mitigate TapTrap.

In addition to exploiting opacity, TapTrap can also confuse users by leveraging extreme "zoom" effects during activity transition animations. To address this, we propose limiting the zoom factor of such animations. Specifically, we suggest setting the zoom factor limit to 400%, as we believe this value balances the aesthetic and functional needs of legitimate animations while reducing the risk of user confusion.

We have disclosed our findings to Google and the affected browser vendors. Firefox and Chrome have implemented mitigations based on the `onEnterAnimationComplete` method, as per our recommendation. Nevertheless, even the latest Android version (Android 15), as of June 2025, lacks system-level mitigations against TapTrap and remains vulnerable.

5 Detecting Malicious Apps in The Wild

To assess whether TapTrap is being actively exploited in the wild, we developed an automated tool to systematically analyze Android apps for potentially malicious animations. Using this tool, we conducted a large-scale analysis of 99,705 Android apps from the Google Play Store.

5.1 Dataset

To compile a comprehensive dataset of apps, we adopted a methodology similar to Steinböck et al. [67]. First, we used `google-play-scraper` [40] to query the package names of the top-ranked free, paid, and grossing apps across all categories from the Austrian Play Store. Subsequently, we recursively gathered the package names of their related apps until we did not identify new package names. This process, performed on December 16, 2024, yielded 123,765 unique package names.

Next, we retrieved metadata for each package name and successfully obtained information for 123,581 apps. We filtered this set to include only apps that are free to download, resulting in 101,672 package names. Using a modified version of `apkeep` [39], configured to emulate a Pixel 6a running Android 15, we downloaded the corresponding APKs. This step, performed between December 18, 2024 and January 2, 2025, yielded 99,722 successfully downloaded apps.

As many apps on the Play Store are distributed as split APKs [8], we employed `APKEditor` [38] to merge these into single APK files. This merging process failed for 17 apps, leaving a final dataset of 99,705 Android apps.

5.2 Evaluation Methodology

We leverage two key properties of Android activity transitions to evaluate the presence of apps that exploit the TapTrap vulnerability. First, only *tween animations* [7] can be used during activity transitions, making them the sole type relevant

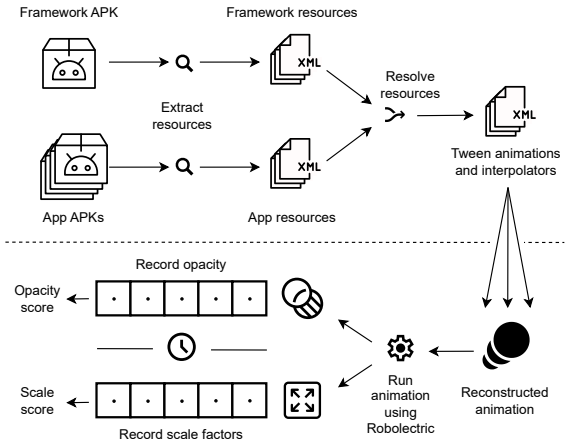


Figure 3: Detection of malicious apps exploiting TapTrap.

to TapTrap. These animations must always be declared as XML files within an app's resources. Second, resource files, including tween animations, must be present at compile time and are immutable at runtime.[2] Consequently, analyzing an app's resource files in the APK ensures that, should TapTrap be exploited in an app, our tool will detect it.

Our methodology comprises two main phases. First, we extract all tween animations declared in the apps' resource files. Second, we analyze each animation to identify behaviors indicative of TapTrap. Fig. 3 illustrates the analysis workflow.

5.2.1 Extracting Animations

An app's resources, including its tween animations, are stored in a binary format compressed in the app's APK. We first decompress and decode the APK using Apktool [24]. To get all candidates for tween animations, we extract all XML files in the app resource directory and filter for those that only contain XML tags used in tween animations.

Android resources often reference other resources using the `@` symbol (e.g., strings or integers) or theme attributes with the `?` symbol, which may vary based on the app's theme. These references may point to resources within the app or external sources, such as the Android framework. To resolve references to Android framework resources, we extract definitions from the `framework-res.apk` of a Pixel 6a running Android 15. We then recursively resolve all references to construct the final animation. As specific resources or theme attributes may vary depending on factors such as screen size, we consider all possible values during this resolution process to ensure we do not overlook any potentially malicious animations. A single XML animation may thus correspond to multiple distinct animations, which are all included in our analysis.

[2]While modifying resources at runtime is technically possible using Runtime Resource Overlays (RRO) [22], this requires elevated privileges and is excluded from our threat model.

5.2.2 Analyzing Animations

After we have gathered all animations, we analyze them for behavior that could potentially be abused for TapTrap. We take each animation as a black box, execute it in a controlled environment, and observe its behavior over time. This approach has multiple advantages over statically parsing the XML files. First, Android animations can be highly complex, often comprising multiple nested animations that interact with one another. Accurately reconstructing the behavior of such animations through static analysis would require an exhaustive and precise model of all potential interactions. This process is error-prone, as it may overlook edge cases, and the Android source code remains the ultimate source of truth for animation behavior. Second, animations can register interpolators that modify the animations behavior over time, such as slowing it down or speeding it up. Statically modeling all possible interpolators and their effects would require significant effort and still risk inaccuracies due to unforeseen interactions. As the `android.view.animation` package, responsible for handling animations, relies on native code, we cannot run animations directly in the JVM. Instead, we leverage Robolectric [41], a testing framework for Android, with which we can achieve this goal without requiring an emulator or physical device. We prepare each animation as the Android framework would, e.g., restricting the duration to 3,000 milliseconds and setting appropriate sizes for the views involved. During execution, we record the view's opacity values (α_i) and its transformation matrices (M_i) at each time interval (i) throughout the animation's lifecycle.

5.2.3 Quantifying Maliciousness

Animations that can be abused for TapTrap exhibit one of two properties: (1) a low opacity over a longer time or (2) a large scale factor ("zoom") over a longer time. We assign each animation two scores that quantify its potential for malicious behavior: a maliciousness score $S_\alpha \in [0,100]$ for abusing opacity and a maliciousness score S_σ for abusing scaling $S_\sigma \in [0,100]$. For the attack to be successful, the animation must exhibit a low opacity or a high scale factor at the beginning of the animation. It should maintain this low opacity or high scale factor for a significant portion of its duration. However, this becomes less important as the animation progresses, as the attacker can repeat the attack before the animation ends.

To capture these requirements, we introduce a weighted scoring function $\phi(i)$ over time $i \in [0,3000]$ milliseconds. We use a logistic-shaped function that begins with slow decay, becomes steeper, and then tapers off, thereby placing greater emphasis on the early part of the animation. We choose an inflection point at 1,500 milliseconds:

$$\phi(i) = 1 - \frac{1}{1 + e^{-0.005(i-1500)}}$$

We normalize the weights to ensure the score remains in the range $[0, 100]$ and define the normalized scoring function $u(i)$:

$$u(i) = \frac{\phi(i)}{\sum_{j=0}^{3000} \phi(j)}$$

In case we observe that an animation exceeds 3,000 milliseconds, we flag it separately, as this indicates that the animation is able to exceed the duration set by the system.

Opacity Score S_α. To compute the opacity score, we consider the opacity α_i at each timestamp $i \in [0, 3000]$. We use an indicator function $\mathbf{1}[0 < \alpha_i \leq 0.1]$, which is 1 if the opacity at time i is greater than 0 and at most 0.1, and 0 otherwise. Note that an opacity of 0 cannot be used for TapTrap, as the view would not receive any input. We conservatively select a threshold of 0.1. Even though an opacity of 0.1 would already be visible to the human eye, we choose this threshold to ensure we do not miss any potential TapTrap attacks.

$$S_\alpha = 100 \cdot \sum_{i=0}^{3000} u(i) \cdot \mathbf{1}[0 < \alpha_i \leq 0.1]$$

Scale Score S_σ. For scale, let σ_{i_x} and σ_{i_y} be the scale factors along the x- and y-dimensions at time i based on the transformation matrix M_i. We use the indicator $\mathbf{1}[\sigma_{i_x} \geq 4 \vee \sigma_{i_y} \geq 4]$, which is 1 if the scale in either dimension exceeds the threshold of at least 4 at time i, and 0 otherwise. We choose a threshold of 4 as the scale factor as, in order to perform TapTrap stealthily using a scale animation, the scale factor must be chosen high enough to distort what the user sees.

$$S_\sigma = 100 \cdot \sum_{i=0}^{3000} u(i) \cdot \mathbf{1}[\sigma_{i_x} \geq 4 \vee \sigma_{i_y} \geq 4]$$

5.3 Results

We successfully extracted animations for 99,018 apps in our dataset, resulting in 2,504,547 animations, of which 55,825 were unique. We define an animation as unique if the MD5 hash of its resolved XML representation, after resolving all references, matches no other hash in the dataset. 46 animations could not be analyzed, primarily due to missing references. We validated our tool against our proof of concepts, and our pipeline successfully assigned appropriate scores.

5.3.1 Finding #1: Animations Can Exceed Duration

In our dataset, we identified 61 unique animations that exceeded the intended duration limit of 3,000 milliseconds. Further analysis revealed that this behavior is due to an off-by-one error in the implementation of the duration restriction in Android's `Animation` class [3]. This bug allows animations to run for up to 6 seconds instead of the intended 3 seconds.

The assigned scores indicate that none of them exhibit behavior that could be exploited for TapTrap. However, the identified off-by-one error increases the attack window to execute TapTrap, thus lowering the barrier to successful exploitation.

5.3.2 Finding #2: TapTrap Not Exploited in the Wild

In our dataset, 22 apps contain animations with an alpha score $S_\alpha \geq 50$, while 6 apps contained those with a scale score $S_\sigma \geq 50$, covering a total of 28 apps. We manually analyzed the animations and reverse-engineered the corresponding apps to evaluate whether these animations were used to perform TapTrap. Fortunately, our analysis found that none of these animations were exploited for TapTrap. Thus, among the extensive dataset analyzed, no evidence suggests that TapTrap is currently being exploited in the wild.

6 Evaluating App Vulnerability

We previously demonstrated that TapTrap can exploit vulnerabilities in both the Android system through bypassing system-level permissions and the broader Web ecosystem through the browser. Beyond these targets, TapTrap also poses a significant threat to the security of individual third-party apps. In this section, we present a large-scale evaluation of 99,705 Android apps to assess their vulnerability to TapTrap. This analysis builds upon the dataset introduced in Sec. 5.1.

6.1 Methodology

As TapTrap relies on custom activity transition animations, which can only be set for same-task transitions, not every app is vulnerable to TapTrap. We define an activity A as vulnerable if it meets all of the following criteria:

- ❶ **Externally Launchable:** Another app can launch A.
- ❷ **Same-Task Launchable:** A can be launched into the same task as any other app.
- ❸ **No Entry Animation Override:** A does not override the custom animation it may be launched with.
- ❹ **No Animation Finished Wait:** A does not wait for the animation to complete before handling user input.

An app is considered vulnerable if it includes a vulnerable activity. We build our analysis on Androguard [37], a popular open-source static analysis tool for Android apps.

6.1.1 Manifest Analysis

Our analysis begins with the `AndroidManifest.xml` file, which declares an app's components. We parse the manifest to check criteria ❶ and ❷. For criterion ❶, we verify the `exported` and `enabled` attributes to confirm that an activity is externally launchable and check for any declared permissions required

to open it. For criterion ❷, we examine the `launchMode` attribute. Activities that specify `standard` or `singleTop` can be launched same-task. Although the manifest can also be used to declare enter or exit animations, our experiments show that animations passed to `startActivity` via an `ActivityOptions` bundle override any manifest-specified animations. Thus, we do not consider manifest-declared animations in our analysis.

6.1.2 Code Analysis

After identifying candidate activities from the manifest, we examine their bytecode to verify criteria ❸ and ❹. We first construct a call graph of the app. For criterion ❸, we then search for calls to `Activity.overridePendingTransition`, which can be used to override the entry animation, and trace the call graph back to an enter method of the activity. If we find a call to `overridePendingTransition`, we flag the activity as overwriting the entry animation. If, however, this call is made after a call to `startActivity` or related methods used to start activities, we discard the method occurrence as it affects a subsequently launched activity rather than the current one. We do not consider alternative methods like `Activity.overrideActivityTransition` or `Window.setAnimations` because they do not override the animation passed via `ActivityOptions` and are not applicable.

For criterion ❹, we check if the activity overrides the `onEnterAnimationComplete` callback. This function is invoked once the enter animation finishes. If an activity overrides this function, we conservatively assume it waits until the animation completes before handling user input. As we show later, only a negligible portion of activities and apps override this function, therefore this conservative approach has minimal impact on the accuracy of our results.

6.1.3 Validation and Limitations

We validated our analysis by manually analyzing a subset of 10 randomly selected apps from our dataset. For each app, we randomly selected 2 activities and attempted to run TapTrap on it. If we were able to run TapTrap on the activity, we classified it as vulnerable. Out of the 20 activities analyzed, we found that our analysis correctly classified all of them.

We believe that this lightweight analysis provides a comprehensive overview of the volume of vulnerable apps. However, certain limitations arise from the inherent constraints of static analysis and specific assumptions in our methodology.

Static Analysis. Detecting activities that override entry animations via `overridePendingTransition` depends on the accuracy of the call graph constructed by Androguard. An inaccurate call graph, especially in the presence of obfuscation techniques, such as reflection, may result in missed calls, leading to false positives. Additionally, we conservatively assume that activities overriding `onEnterAnimationComplete` wait for animations to finish before processing user input, which could cause false negatives. However, as only 0.1% of activities override this function, it minimally affects our results.

Activity/App Property	Activities	Apps
Externally launchable	10.1%	99,278 (99.7%)
Same-task launchable	93.6%	98,478 (98.9%)
Restricts animations	5.7%	37,017 (37.2%)
Waits for animation end	0.1%	599 (0.6%)
Total vulnerable	6.8%	76,035 (76.3%)

Table 3: Activities and apps meeting the vulnerability criteria. An app meets a criterion if it includes an activity satisfying it.

Threat Model Simplifications. Our analysis treats each activity as independent and may overlook complex app flows. For example, an exported activity classified as vulnerable may only launch other non-exported activities in new tasks, leading to an overestimation of vulnerabilities. Furthermore, labeling an activity as vulnerable does not imply it is easily exploitable. Real-world UI flows or multi-step interactions, such as clicks or swipes, can reduce TapTrap's practical risk. Additionally, a vulnerable activity may not be security-sensitive, e.g., a splash screen may be vulnerable but pose little actual risk.

6.2 Results

Our analysis successfully processed 99,612 apps (99.9%). On average, it took 111 seconds per app, with a timeout of 1 hour reached for 6 apps. In total, the apps in our dataset contained 3,699,536 activities. Table 3 summarizes the findings.

❶ **Externally Launchable.** We found that 99.5% of activities do not require permissions to be launched. Moreover, 382,089 (10.3%) are exported, and 3,688,165 (99.7%) are enabled, resulting in 372,972 externally launchable activities (10.1%).

❷ **Same-Task Launchable.** The majority of activities (93.6%) can be launched in the same task, as they use a `launchMode` of either `standard` or `singleTop`.

❸ **Overriding Entry Animations.** Overriding entry animations is uncommon; only 209,850 activities (5.7%) override entry animations using `overridePendingTransition`, thereby preventing attackers from supplying custom animations.

❹ **Waiting for Enter Animations to Complete.** Similarly, only 3,785 activities (0.1%) override the `onEnterAnimationComplete` callback, indicating that a negligible amount of activities have the capability to wait for animations to complete before processing user input.

App Vulnerability. In total, 252,776 activities (6.8%) meet all four criteria for vulnerability. This corresponds to 76,035 apps that contain a vulnerable activity and that we therefore classify as vulnerable. With over 76% of the analyzed apps being vulnerable, these findings underscore that TapTrap not only targets Android system components, but also poses a significant threat to the broader mobile app landscape.

7 Evaluating User Awareness

We conducted a user study to evaluate the real-world practicality of TapTrap and users' ability to detect TapTrap-based attacks during typical mobile interactions. This section details the recruitment process, study design, experiment setup, and discusses the study's key findings.

7.1 Recruitment

We recruited 20 participants through various channels, including word of mouth among students, lecture hall announcements, and outreach to administrative staff within our faculty. All participants were at least 18 years old, with 14 participants aged 24 or younger and 2 participants older than 34. As compensation, they received hot and cold beverages. We did not restrict participation based on prior experience with Android devices. Nevertheless, 14 participants reported having used an Android device within the last 5 years.

7.2 Experiment Design

We initially provided participants with only minimal information about the study's objectives. To avoid introducing bias, we framed the study as a general evaluation of user interaction with apps without disclosing the real purpose of the study.

7.2.1 Testing App: KillTheBugs

The study centered on the *KillTheBugs* game we developed. Participants advanced through three levels by squishing bugs that appeared on the screen and played the game on a Pixel 6a device running Android 15 that we provided. Each level included a different attack scenario. The game is discussed in more detail in Appendix B and included in the paper artifacts.

Level 1. TapTrap is used to open a malicious website in a Custom Tab that requests the user's location.

Level 2. Similar to Level 1, but the website requests camera permissions. To disguise the privacy indicator displayed on our Pixel 6a device when the camera is accessed, we match the app background color to the indicator's background color.

Level 3. This level escalates privileges by requesting device admin permissions and locks the screen once granted.

7.2.2 Organization

The study was divided into the following phases. The content of the questionnaires is provided in Appendix B.

Initial Questionnaire. Participants began the experiment by completing a brief questionnaire about themselves.

First Run. We instructed participants to play the game and told them to assume that they recently downloaded it from the Google Play Store, ensuring no permissions had been granted to the app. The primary objective of this phase was to assess participants' ability to detect TapTrap without prior knowledge of the vulnerability.

Intermediate Questionnaire. After the first gameplay session, participants completed a questionnaire to capture their general impressions using free-form questions.

Intermediate Debriefing. We informed them that we are conducting a security study, and the app tried to lure them into stealthily granting specific permissions.

Second Run. Participants replayed the game. This phase assessed whether TapTrap remains stealthy enough to evade detection, even when users are aware of potential threats.

Final Questionnaire. Following this second session, they completed a questionnaire to capture their observations.

Debriefing. The study concluded with a final debriefing session, where participants could play the game a third time. During this session, we adjusted the animations to make the attacks noticeable to the participants (as shown in Fig. 4c). This ensured that participants left the study with a clear understanding of the attack and did not feel tricked.

7.3 Results

This section presents the findings of the user study. We also discuss the key insights derived from these findings.

7.3.1 Uninformed Users

After the first session, participants described the game as "repetitive and slow-paced" and "wondered when the game would end". Two participants noticed minor glitches, such as a quick flash on the screen, and one participant observed that they sometimes needed to tap twice on a bug. However, no one initially linked these glitches to any malicious behavior.

Level 1. None of the participants detected any anomalies in this level beyond the general issues mentioned above.

Level 2. Four participants (21%) noticed the camera indicator in the status bar. One participant noted, "The app didn't ask for permissions, but I saw the camera icon blink, which seemed odd." while another said, "There was a camera icon in the top row as if the camera was accessed". The remaining participants did not notice anything abnormal. For one participant, Level 2 failed as they did not interact with the game within the TapTrap attack window of 6 seconds.

Level 3. Four participants noted that the screen suddenly went black. They speculated that the "app broke down", that "maybe it crashed", and that "the crash was interesting". One noted that a pop-up appeared before the screen went black.

7.3.2 Informed Users

Most participants tried to be more cautious in the second run. However, one participant was so cautious that they did not

grant any permissions during the second session as they did not interact with the game within the attack window. Another participant was too cautious in Level 1 and was too slow to grant the location permission.

Level 1. No participant detected the attack during this level.

Level 2. 14 participants (74%) noticed the camera indicator.

Level 3. Three participants observed anomalies before the screen locked. Two participants reported briefly seeing a permission prompt, while one mentioned noticing some text flashing on the screen just before the lock occurred.

7.3.3 Key Insights

Our results indicate that, without visible security indicators, uninformed users fail to detect TapTrap and that such attacks can be performed completely stealthily. Even with security indicators present, as in Level 2, only 21% of participants noticed such indicators. This suggests that security indicators are easily overlooked, especially when an app wants to masquerade them and they are not expected by the user.

Alarmingly, the results suggest that even vigilant users who are informed about the app's malicious behavior struggle to detect the attack when no security indicators are present. While the detection rate of the camera indicator significantly increased for informed users from 21% to 74%, attacks that do not trigger an indicator went largely unnoticed.

8 Related Work

Previous work can be divided into previous attacks on the mobile GUI, their mitigations, and clickjacking attacks.

Attacks on the Mobile GUI. Attacks that target the mobile GUI have received various attention in the research community. In Sec. 4 we have already extensively discussed previous work that relies on tapjacking techniques [2, 31, 36, 49, 50, 53, 59, 70, 71, 74]. Bianchi et al. [28] investigated deception techniques in the Android UI. Through automated state exploration of Android platform APIs, the authors identified novel attack vectors on the Android GUI, such as inescapable fullscreen overlays, and categorized existing techniques, including phishing-style and tapjacking-style attacks. Bove [29] conducted a systematization of knowledge on the evolution of Trusted UI on mobile devices. Similarly, Bove and Kalysch [30] explored the evolution of UI-based attack vectors on Android. Other previous works rely on task hijacking or app hijacking in which a malicious app inserts itself into the foreground to perform phishing attacks [28, 33, 62, 69].

Tapjacking Prevention. Meanwhile, an increasing number of works also propose defenses against tapjacking attacks. Bianchi et al. [28] proposed modifications to the Android OS to add an identification of what app is currently being displayed on top. Fernandes et al. [35] analyzed the proposed mechanism and found that this creates a side-channel and proposed another defense mechanism that prevents non-system background apps from creating overlays. Ren et al. [61] proposed WindowGuard, a defense mechanism that relays whether an overlay should be displayed in security-critical scenarios to the user. Possemato et al. [60] developed ClickShield, an OS-level mechanism that evaluates the visual uniformity of overlays on the screen to differentiate between benign and malicious overlays. Most recently, Yan et al. [73] and Gong et al. [42] introduced OverlayChecker, a system for the early detection of overlay-based malware during the app market review process.

Web Clickjacking. The concept of clickjacking, the Web counterpart to tapjacking, was first introduced in a 2002 Firefox Bug report [58], though the term "clickjacking" originated in 2008 in a blog post by Grossman and Hansen [45]. In 2010, Rydstedt et al. [63] analyzed the deployment of clickjacking defenses across the top 500 websites, marking an early effort to understand its mitigation. Shortly after, Balduzzi et al. [25] developed an automated tool to detect clickjacking attacks and conducted a large-scale study to assess their prevalence. In 2012, Lekies et al. [52] identified limitations in existing defenses and evaluated the adoption of prevention mechanisms on a broader scale. Around the same time, Huang et al. [48] introduced new attack variants and proposed an opt-in defense mechanism to ensure that sensitive UI elements remain visible to users. Further research explored the role of human perception in clickjacking attacks. Akhawe et al. [1] proposed five novel attack techniques. Most recently, in 2020, Calzavara et al. [32] examined inconsistencies in browser support for framing control mechanisms. They introduced a formal framework for automated analysis and conducted a large-scale study to identify these inconsistencies.

9 Conclusion

In this work, we introduced TapTrap, a novel tapjacking attack that leverages activity transition animations to create a mismatch between user perception and the underlying state of apps. TapTrap bypasses existing mitigations at both the system and application levels, enabling a malicious app to circumvent sensitive permissions, such as access to the camera, location, and notifications, and escalate its impact to critical actions, including full device erasure.

We demonstrated that TapTrap extends beyond the Android ecosystem, posing a significant threat to apps and websites as well. Our analysis of 99,705 apps from the Google Play Store revealed no evidence of TapTrap being exploited in the wild. However, further analysis showed that 76.3% of apps are vulnerable to TapTrap. To assess its practical impact, we conducted a user study with 20 participants. Alarmingly, all participants failed to detect at least one variant of the attack, underscoring its stealthiness and the risks it poses to users.

As TapTrap represents a new class of UI-based attacks that does not rely on traditional overlay techniques, it exposes a fundamental gap in the current security model. As of June 2025, even the latest Android version (Android 15) remains vulnerable, highlighting the urgent need for defenses.

While our analysis focused on Android activity transitions, the broader security impact of animations and user perception remains largely unexplored. Future work could investigate the security implications of animations across different platforms, as they may exhibit similar weaknesses.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback and the participants of our user study for their time and effort. This work was supported by the Vienna Science and Technology Fund (WWTF) and the City of Vienna [Grant ID: 10.47379/ICT22060], the Austrian Science Fund (FWF) [Grant ID: 10.55776/F8515-N], and SBA Research (SBA-K1 NGC), a COMET Center within the COMET – Competence Centers for Excellent Technologies Programme and funded by BMIMI, BMWET, and the federal state of Vienna. The COMET Programme is managed by FFG. The paper includes icons from Icons8, Font Awesome, and Material Icons, as well as by Donnnno (GPL, unmodified), Shannon E. Thomas (CC BY, unmodified), Coreui (GPL, unmodified), Catalin Fertu (CC BY, unmodified), and VMware (MIT). Full license texts are provided in the paper artifact. Analyses in this paper made use of GNU Parallel [68], a parallel execution tool.

Open Science

To foster future research and reproducibility, we make the following artifacts of this paper publicly available:

- **TapTrap PoC:** A proof-of-concept implementation demonstrating the TapTrap attack (Sec. 3).
- **Dataset Preparation Pipeline:** Scripts for crawling the Google Play Store, downloading apps, and preparing them for analysis (Sec. 5.1).
- **Malicious App Detection Pipeline:** Code and results used to identify malicious apps (Sec. 5).
- **Vulnerable App Detection Pipeline:** Code and results used to identify vulnerable apps (Sec. 6).
- **User Study Materials:** Materials used in the user study, including the information sheet, consent forms, questionnaires, app, and study website (Sec. 7).

The artifacts are available at `https://doi.org/10.5281/zenodo.15519676`. Due to the considerable size of the APK dataset, we provide access to it upon request. Additionally, we provide further material and up-to-date information on TapTrap at `https://taptrap.click`.

Ethics Considerations

User Study

Our institution's ethics committee reviewed the user study conducted in this work. We outline the core concerns of the study and the measures we took to mitigate them.

Recruitment Partiality. Participants were, among other channels, recruited from courses offered at our institution. To mitigate the potential concern that users might feel obligated to participate, we emphasized that participation was entirely voluntary and provided no academic advantage. Additionally, recruitment was conducted by two authors of this paper who were not affiliated with any of the courses where recruitment took place, to ensure impartiality and minimize any perceived pressure to participate.

Participant's Feelings of Deception. Participants who do not identify the attack during the study might feel deceived or confused. To mitigate this, we included a final debriefing session where participants played the game a third time. During this session, we explicitly demonstrated the attack mechanism and explained where and how the attacks occurred. This ensured that participants left the study with a clear understanding of the attack and its implications.

Responsible Disclosure

In October 2024, we reported the TapTrap attack to the Android Security Team. The report was intially acknowledged but later marked as a duplicate of an existing issue, to which we were not granted access to. The security team stated that the vulnerability would be addressed in a future release, but provided us with no timeline or further information in this regard. As of June 2025, the vulnerability remains unpatched.[3]

We also disclosed the vulnerability to the affected browser vendors, including Google Chrome, Mozilla Firefox, Microsoft Edge, and Brave. Both Edge and Brave acknowledged the issue but deferred responsibility to Chrome, as their browsers are based on the Chromium engine. Chrome fixed the issue in version 135, assigned CVE-2025-3067, and awarded us a bug bounty of $10K, while Mozilla addressed it in Firefox 136 with CVE-2025-1939. Both browser vendors followed our recommendations and mitigated the attack by relying on the `onEnterAnimationComplete` method.

In November 2024, we separately reported the off-by-one error in the animation duration restriction to the Android Security Team. This report was marked as "WontFix', explaining that it "does not affect the security of the Android platform". While the issue may not be critical in isolation, it effectively doubles the attack window for TapTrap, thereby increasing the likelihood of a successful attack.

[3]We tested TapTrap based on the PoC included in the paper artifacts on an up-to-date Pixel 8a (Android 15, last security update May 5, 2025, build `BP1A.250505.005.B1`).

References

[1] Devdatta Akhawe, Warren He, Zhiwei Li, Reza Moazzezi, and Dawn Song. Clickjacking Revisited: A Perceptual View of UI Security. In *Workshop on Offensive Technologies (WOOT)*, 2014.

[2] Efthimios Alepis and Constantinos Patsakis. Trapped by the UI: The Android Case. In *Research in Attacks, Intrusions, and Defenses (RAID)*. Springer, 2017.

[3] Android Code Search. Animation.java. https://cs.android.com/android/platform/superproject/main/+/main:frameworks/base/core/java/android/view/animation/Animation.java%3Bl=490, December 2024. [Online; accessed December 12, 2024].

[4] Android Code Search. Commit 3d9d177. https://cs.android.com/android/_/android/platform/frameworks/base/+/3d9d17788fb713787bda740a335304d1b346e19b, January 2024. [Online; accessed January 20, 2025].

[5] Android Developers. Android 8.0 Behavior Changes. https://developer.android.com/about/versions/oreo/android-8.0-changes, May 2024. [Online; accessed January 22, 2025].

[6] Android Developers. Android Protected Confirmation. https://developer.android.com/privacy-and-security/security-android-protected-confirmation, November 2024. [Online; accessed December 2, 2024].

[7] Android Developers. Animation resources. https://developer.android.com/guide/topics/resources/animation-resource, January 2024. [Online; accessed November 29, 2024].

[8] Android Developers. Build multiple APKs. https://developer.android.com/build/configure-apk-splits, May 2024. [Online; accessed December 5, 2024].

[9] Android Developers. Device administration overview. https://developer.android.com/work/device-admin, December 2024. [Online; accessed December 9, 2024].

[10] Android Developers. Intents and intent filters. https://developer.android.com/guide/components/intents-filters, December 2024. [Online; accessed December 11, 2024].

[11] Android Developers. Introduction to activities. https://developer.android.com/guide/components/activities/intro-activities, May 2024. [Online; accessed January 2, 2025].

[12] Android Developers. MotionEvent. https://developer.android.com/reference/android/view/MotionEvent#FLAG_WINDOW_IS_OBSCURED, July 2024. [Online; accessed December 2, 2024].

[13] Android Developers. Permissions on Android. https://developer.android.com/guide/topics/permissions/overview, December 2024. [Online; accessed December 11, 2024].

[14] Android Developers. Tapjacking. https://developer.android.com/privacy-and-security/risks/tapjacking, September 2024. [Online; accessed December 2, 2024].

[15] Android Developers. Tasks and the back stack. https://developer.android.com/guide/components/activities/tasks-and-back-stack, March 2024. [Online; accessed November 29, 2024].

[16] Android Developers. Toast. https://developer.android.com/reference/android/widget/Toast, December 2024. [Online; accessed January 2, 2024].

[17] Android Developers. Toasts overview. https://developer.android.com/guide/topics/ui/notifiers/toasts, January 2024. [Online; accessed January 25, 2025].

[18] Android Developers. View. https://developer.android.com/reference/android/view/View#setFilterTouchesWhenObscured(boolean), July 2024. [Online; accessed December 2, 2024].

[19] Android Developers. WindowManager.LayoutParams. https://developer.android.com/reference/android/view/WindowManager.LayoutParams#FLAG_NOT_TOUCHABLE, December 2024. [Online; accessed January 3, 2024].

[20] Android Developers. Restrictions on starting activities from the background. https://developer.android.com/guide/components/activities/background-starts, May 2025. [Online; accessed May 20, 2025].

[21] Android Open Source Project. Application Sandbox. https://source.android.com/docs/security/app-sandbox, November 2024. [Online; accessed December 11, 2024].

[22] Android Open Source Project. Change the value of an app's resources at runtime. https://source.android.com/docs/core/runtime/rros, November 2024. [Online; accessed December 11, 2024].

[23] Android Open Source Project. Privacy indicators. https://source.android.com/docs/core/

permissions/privacy-indicators, November 2024. [Online; accessed December 2, 2024].

[24] Apktool. Apktool. https://apktool.org. [Online; accessed November 30, 2024].

[25] Marco Balduzzi, Manuel Egele, Engin Kirda, Davide Balzarotti, and Christopher Kruegel. A Solution for the Automated Detection of Clickjacking Attacks. In *Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM, 2010.

[26] Philipp Beer, Marco Squarcina, Lorenzo Veronese, and Martina Lindorfer. Tabbed Out: Subverting the Android Custom Tab Security Model. In *Symposium on Security and Privacy (S&P)*. IEEE, 2024.

[27] Philipp Beer, Lorenzo Veronese, Marco Squarcina, and Martina Lindorfer. The Bridge between Web Applications and Mobile Platforms is Still Broken. In *Workshop on Designing Security for the Web (SecWeb)*. IEEE, 2022.

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

[29] Davide Bove. SoK: The Evolution of Trusted UI on Mobile. In *Asia Conference on Computer and Communications Security (ASIACCS)*. ACM, 2022.

[30] Davide Bove and Anatoli Kalysch. In pursuit of a secure UI: The cycle of breaking and fixing Android's UI. *it-Information Technology*, 2019.

[31] Yan Cai, Yutian Tang, Haicheng Li, Le Yu, Hao Zhou, Xiapu Luo, Liang He, and Purui Su. Resource Race Attacks on Android. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020.

[32] Stefano Calzavara, Sebastian Roth, Alvise Rabitti, Michael Backes, and Ben Stock. A Tale of Two Headers: A Formal Analysis of Inconsistent Click-Jacking Protection on the Web. In *USENIX Security Symposium (USENIX)*, 2020.

[33] Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *USENIX Security Symposium (USENIX)*, 2014.

[34] Chrome Developers. Overview of Android Custom Tabs. https://developer.chrome.com/docs/android/custom-tabs, February 2020. [Online; accessed Jan 17, 2025].

[35] Earlence Fernandes, Qi Alfred Chen, Justin Paupore, Georg Essl, J Alex Halderman, Z Morley Mao, and Atul Prakash. Android UI Deception Revisited: Attacks and Defenses. In *Financial Cryptography and Data Security (FC)*. Springer, 2017.

[36] Yanick Fratantonio, Chenxiong Qian, Simon P. Chung, and Wenke Lee. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In *Symposium on Security and Privacy (S&P)*. IEEE, 2017.

[37] GitHub. androguard. https://github.com/androguard/androguard. [Online; accessed December 5, 2024].

[38] GitHub. APKEditor. https://github.com/REAndroid/APKEditor. [Online; accessed December 5, 2024].

[39] GitHub. apkeep. https://github.com/EFForg/apkeep. [Online; accessed December 5, 2024].

[40] GitHub. google-play-scraper. https://github.com/JoMingyu/google-play-scraper. [Online; accessed December 5, 2024].

[41] GitHub. robolectric. https://github.com/robolectric/robolectric. [Online; accessed December 11, 2024].

[42] Liangyi Gong, Zhenhua Li, Hongyi Wang, Hao Lin, Xiaobo Ma, and Yunhao Liu. Overlay-Based Android Malware Detection at Market Scales: Systematically Adapting to the New Technological Landscape. *IEEE Transactions on Mobile Computing*, 2022.

[43] Google. Manage permissions from the privacy dashboard. https://support.google.com/android/answer/13530434, January 2025. [Online; accessed January 7, 2025].

[44] Google for Developers. Device admin deprecation. https://developers.google.com/android/work/device-admin-deprecation, October 2024. [Online; accessed January 2, 2025].

[45] Robert Hansen and Jeremiah Grossman. Clickjacking. https://web.archive.org/web/20091209043515/http://www.sectheory.com/clickjacking.htm, December 2008. [Online; accessed January 2, 2024].

[46] Duncan Hodges and Oliver Buckley. Reconstructing What You Said: Text Inference Using Smartphone Motion. *IEEE Transactions on Mobile Computing*, 2018.

[47] HTTP Archive. Web Almanac. https://almanac.httparchive.org/en/2024/, 2024. [Online; accessed January 10, 2024].

[48] Lin-Shung Huang, Alex Moshchuk, Helen J. Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and Defenses. In *USENIX Security Symposium (USENIX)*, 2012.

[49] Ken Johnson. Revisiting Android TapJacking. https://web.archive.org/web/20171121203845/https://nvisium.com/blog/2011/05/26/revisiting-android-tapjacking, May 2011. [Online; accessed 3. Dec. 2024].

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

[51] Soheil Khodayari and Giancarlo Pellegrino. The State of the SameSite: Studying the Usage, Effectiveness, and Adequacy of SameSite Cookies. In *Symposium on Security and Privacy (S&P)*. IEEE, 2022.

[52] Sebastian Lekies, Mario Heiderich, Dennis Appelt, Thorsten Holz, and Martin Johns. On the Fragility and Limitations of Current Browser-Provided Clickjacking Protection Schemes. In *Workshop on Offensive Technologies (WOOT)*, 2012.

[53] Tongbo Luo, Xing Jin, Ajai Ananthanarayanan, and Wenliang Du. Touchjacking Attacks on Web in Android, iOS, and Windows Phone. In *International Symposium on Foundations and Practice of Security (FPS)*. Springer, 2012.

[54] MDN Web Docs. CSP: frame-ancestors. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors, August 2024. [Online; accessed January 17, 2025].

[55] MDN Web Docs. Permissions API. https://developer.mozilla.org/en-US/docs/Web/API/Permissions_API, November 2024. [Online; accessed January 21, 2025].

[56] MDN Web Docs. X-Frame-Options. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options, December 2024. [Online; accessed January 17, 2025].

[57] Rowan Merewood. SameSite cookies explained. https://web.dev/articles/samesite-cookies-explained, May 2019. [Online; accessed January 17, 2025].

[58] Mozilla Bugzilla. Bug 154957 - iframe content background defaults to transparent. https://bugzilla.mozilla.org/show_bug.cgi?id=154957, June 2002. [Online; accessed December 19, 2024].

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

[60] Andrea Possemato, Andrea Lanzi, Simon Pak Ho Chung, Wenke Lee, and Yanick Fratantonio. ClickShield: Are You Hiding Something? Towards Eradicating Clickjacking on Android. In *Conference on Computer and Communications Security (CCS)*. ACM, 2018.

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

[62] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. Towards Discovering and Understanding Task Hijacking in Android. In *USENIX Security Symposium (USENIX)*, 2015.

[63] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites. *IEEE Oakland Web*, 2010.

[64] Amit Kumar Sikder, Giuseppe Petracca, Hidayet Aksu, Trent Jaeger, and A. Selcuk Uluagac. A Survey on Sensor-Based Threats and Attacks to Smart Devices and Applications. *IEEE Communications Surveys & Tutorials*, 2021.

[65] David Silver, Suman Jana, Dan Boneh, Eric Chen, and Collin Jackson. Password Managers: Attacks and Defenses. In *USENIX Security Symposium (USENIX)*, 2014.

[66] StatCounter Global Stats. Mobile Browser Market Share Worldwide. https://gs.statcounter.com/browser-market-share/mobile/worldwide, December 2024. [Online; accessed December 28, 2024].

[67] Magdalena Steinböck, Jakob Bleier, Mikka Rainer, Tobias Urban, Christine Utz, and Martina Lindorfer. Comparing Apples to Androids: Discovery, Retrieval, and Matching of iOS and Android Apps for Cross-Platform Analyses. In *International Conference on Mining Software Repositories (MSR)*. IEEE/ACM, 2024.

[68] Ole Tange. GNU Parallel 20240122. https://doi.org/10.5281/zenodo.463844, 2024.

[69] Güliz Seray Tuncay, Jingyu Qian, and Carl A. Gunter. See No Evil: Phishing for Permissions with False Transparency. In *USENIX Security Symposium (USENIX)*, 2020.

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

[71] Xianbo Wang, Shangcheng Shi, Yikang Chen, and Wing Cheong Lau. PHYjacking: Physical Input Hijacking for Zero-Permission Authorization Attacks on Android. In *Network and Distributed System Security Symposium (NDSS)*, 2022.

[72] Zhi Xu, Kun Bai, and Sencun Zhu. TapLogger: Inferring User Inputs On Smartphone Touchscreens Using On-board Motion Sensors. In *Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. ACM, 2012.

[73] Yuxuan Yan, Zhenhua Li, Qi Alfred Chen, Christo Wilson, Tianyin Xu, Ennan Zhai, Yong Li, and Yunhao Liu. Understanding and Detecting Overlay-based Android Malware at Market Scales. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 2019.

[74] Lingyun Ying, Yao Cheng, Yemian Lu, Yacong Gu, Purui Su, and Dengguo Feng. Attacks and Defence on Android Free Floating Windows. In *Asia Conference on Computer and Communications Security (ASIACCS)*. ACM, 2016.

A TapTrap Example Implementation

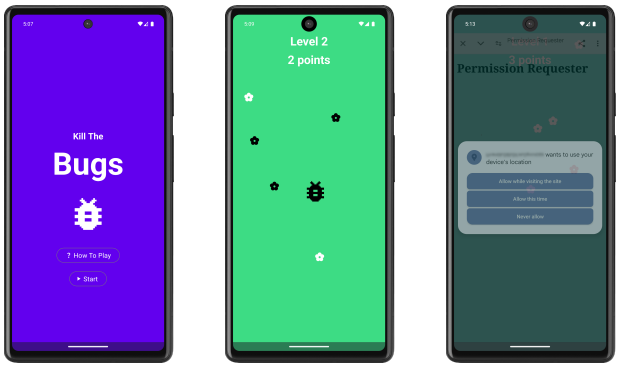
This section outlines a minimal implementation of the TapTrap attack. A full proof-of-concept implementation is included in the artifacts accompanying this paper.

To define the required low-opacity animation, we create a custom tween animation in the `res/anim` directory of the Android project, e.g., `fade_in.xml`, shown in Listing 1.

```xml
<?xml version="1.0" encoding="utf-8"?>
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
    android:fromAlpha="0.01"
    android:toAlpha="0.01"
    android:repeatCount="1"
    android:duration="2999"
/>
```

Listing 1: Low-opacity tween animation (`fade_in.xml`.)

This animation leverages the off-by-one error discussed in Sec. 5, resulting in a total runtime of the animation of 5,998 milliseconds. To apply this animation during an activity transition to `TargetActivity`, we invoke `makeCustomAnimation` and pass the resulting `ActivityOptions` to `startActivity`, as shown in Listing 2.



(a) Start screen of the game (b) Bugs appear on the screen (c) Debriefing mode showing the attack

Figure 4: KillTheBugs user study app.

```kotlin
// Create the Intent
val intent: Intent =
    Intent(this, TargetActivity::class.java)

// Add the animation
val options: ActivityOptions =
    ActivityOptions.makeCustomAnimation(
        this, R.anim.fade_in, 0
    )

// Start the activity
startActivity(intent, options.toBundle())
```

Listing 2: Launching `TargetActivity` with the custom low-opacity TapTrap animation.

B User Study

We provide additional information on the user study we conducted to assess the real-world practicality of TapTrap and users' ability to detect such attacks.

B.1 Testing App: KillTheBugs

Although the *KillTheBugs* game can be adapted for other devices, we optimized it for a Pixel 6a device running Android 15, which was used by all the participants during the study. The user interface of the game can be seen in Fig. 4.

To realistically mimic an attacker's behavior, our objective was to execute the attack as stealthily as possible. A primary challenge stemmed from the system indicators shown when the camera is accessed. On the Pixel 6a, three visual cues are displayed: ❶ a green dot in the top-right corner, ❷ a pulse animation around the front-facing camera cutout, and ❸ a camera icon in the top right corner in the device's status bar.

We bypassed ❶ by setting the app's background color to match that of the green dot. To address ❷, we added a black ring around the camera cutout that hides the pulse animation, which leaves ❸ as the only indicator of camera access.

B.2 Questionnaires

This section contains the questionnaires used in the user study. Further materials used in the study, such as the information sheet, are available in the artifacts accompanying this paper.

Initial Questionnaire

In the initial questionnaire, we collected information about the participants before they started the game.

1. Please select your age group:
   - 18-24
   - 25-34
   - 35-44
   - 45-54
   - 55+
2. Which mobile operating systems are you currently using on your mobile phone(s)? (Select all that apply)
   - Android
   - iOS
   - others (please specify) [*free text*]
   - I don't know
   - I am not using a mobile phone
3. Which mobile operating systems have you used on your mobile phone(s) in the past five years? (Select all that apply)
   - Android
   - iOS
   - others (please specify) [*free text*]
   - I don't know
   - I am not using a mobile phone

Intermediate Questionnaire

Participants completed this questionnaire after the first game session and before we disclosed the study's true purpose.

1. How would you rate your overall experience with the app?
   - Very poor (1)
   - Poor (2)
   - Fair (3)
   - Good (4)
   - Very good (5)

   Please explain why: [*free text*]
2. Did you encounter any issues, difficulties, technical problems, or glitches while using the app?
   - Yes
   - No

   Please explain why: [*free text*]
3. Were there any moments when the app caught your attention?
   - Yes
   - No

   Please explain why: [*free text*]
4. Is there anything else you'd like to share about your experience with the app?

   Please describe if applicable: [*free text*]

Final Questionnaire

We administered the final questionnaire following the second and final gameplay session.

1. Were you able to detect when the vulnerability/vulnerabilities was/were exploited?
   - Yes
   - No

   If yes, please describe when and to what extent: [*free text*]

USENIX Security '25 Artifact Appendix: TapTrap: Animation-Driven Tapjacking on Android

Philipp Beer
TU Wien

Marco Squarcina
TU Wien

Sebastian Roth
University of Bayreuth

Martina Lindorfer
TU Wien

A Artifact Appendix

A.1 Abstract

We provide multiple artifacts to reproduce the results presented in the paper and support future work that builds on our findings. Specifically, we include the following artifacts:

- **Dataset preparation:** Scripts for crawling the Google Play Store, downloading apps, and preparing them for analysis (`/dataset_preparation`). Due to the dataset size, we cannot publicly release the full set of apps used. Reviewers are granted access to it as outlined in Section A.3.1.
- **Malicious app detection:** Code and results for malicious app detection (`/malicious_app_detection`).
- **Vulnerable app detection:** Code and results for vulnerable app detection (`/vulnerable_app_detection`).
- **User study:** Materials from the user study, including the information sheet, consent forms, questionnaires, app, and website used during the study (`/user_study`).
- **TapTrap PoC:** Proof-of-concept of TapTrap (`/poc`).
- **Reproducibility scripts:** Scripts to reproduce the results presented in the paper (`/reproducibility`).
- **Supplementary files:** Additional files not directly relevant for artifact evaluation, such as assets and paper-related licenses (`/assets` and `/paper_licenses`)

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Excessive or rapid scraping and downloads of APKs from the Play Store may violate the Play Store's terms of service and could result in temporary or permanent IP bans. The dataset preparation pipeline uses a conservative approach to avoid excessive downloads. We nevertheless recommend reviewers use a dedicated machine to avoid potential issues.

Running the KillTheBugs app and the TapTrap PoC are intended solely to demonstrate the security vulnerability described in the paper. We do not perform destructive operations and do not collect personal data.

A.2.2 How to access

The artifacts accompanying the paper are hosted at https://github.com/secpriv/taptrap and archived at https://doi.org/10.5281/zenodo.15519676.

A.2.3 Hardware dependencies

Access to the APKs. Due to the size of the APK dataset used in the paper, we cannot distribute it directly. Reviewers are granted access to it as outlined in Section A.3.1.

System Requirements. Both x86 and ARM architectures are supported. For running an Android emulator, however, we recommend a Mac with Apple Silicon. A minimum of 16 GB RAM and 150 GB available disk space are suggested.

Physical Android Device. We recommend running the apps on a physical Pixel 6a device running Android 15. While they can be run on other devices, screen element positioning may differ and require manual adjustment for the attack to run. Alternatively, the app can be executed in an emulator.

A.2.4 Software dependencies

Operating System. We have tested and support Ubuntu 24.04 and macOS 15 with a desktop environment. Other systems *may* require adjustments.

Docker. Install Docker (see `/reproducibility/README.md` for a step-by-step guide or https://docker.com/get-started for official instructions).

Rsync. We require `rsync` (preinstalled on Ubuntu 24.04 and macOS 15) to retrieve the APK dataset. We have, however, experienced issues with it on macOS 15 and suggest installing it via Homebrew instead (`brew install rsync`).

Java. To be able to use the necessary Android dependencies, install a recent version of Java (see https://www.java.com/en/download/manual.jsp)

Android Dependencies. Install the Android dependencies (see `/reproducibility/README.md` for a step-by-step guide):

- Download and install the Android command line tools, which include `sdkmanager`.
- Install the platform tools to install `ADB`.
- Set the `ANDROID_HOME` environment variable.
- Add the platform tools to `$PATH`.

A.2.5 Benchmarks

Evaluating the analyses pipelines requires access to the APK dataset we used. See Section A.3.1 for access instructions.

A.3 Set-up

A.3.1 Installation

Clone the Repository. Clone the artifact repository using `git clone https://github.com/beerphilipp/taptrap.git` in a directory of your choice.

Install the Dependencies. See A.2.4 for instructions.

Obtain a Google AAS token. Downloading apps from the Play Store requires a Google account and an AAS token. We provide such credentials for reviewers to use. Other researchers may create a new Google account and refer to `/dataset_preparation/downloader/README.md` for a summary on how to generate an AAS token.

Download APKs. Our experiments use a subset of 500 randomly selected apps and 266 predefined apps from the dataset. To access the dataset, save the private SSH key that we provide for reviewers to a file named `~/.ssh/taptrap_key` and give it the correct permissions with `chmod 600 ~/.ssh/taptrap_key`. Researchers may request access to the APK dataset by contacting the authors.
Run the following commands in the artifact's root directory:

- Select 500 random apps:

```
rsync -e "ssh -i ~/.ssh/taptrap_key" -azn \
--out-format="%n" dl@download.st1.secpriv.wien: . | \
grep -v "/$" | sort -R | head -n 500 > /tmp/apps.txt
```

- Add the set of predefined apps:

```
cat reproducibility/fixed_apps.txt >> /tmp/apps.txt
```

- Download the selected apps, where `<DIR>` refers to where the APKs should be stored:

```
rsync -e "ssh -i ~/.ssh/taptrap_key" -avxz \
--files-from /tmp/apps.txt \
dl@download.st1.secpriv.wien: <DIR>
```

Start the Android Emulator or Connect a Device. Connect the physical device to the host machine via USB. If you are using an emulator, run `reproducibility/start_emulator.sh` in the repository's root directory to automatically download the correct emulator image for your system and start it.

Set up the Android device. On physical devices, enable USB debugging in *Settings* > *About phone*, then tap *Build number* seven times to enable developer options. Go to *System* > *Developer options* and enable *USB debugging*. This step is not required for emulators.

A.3.2 Basic Test

To verify correct installation and setup, run `reproducibility/basic_test.sh <email> <token>` in the repository's root directory. Replace `<email>` and `<token>` with the Google credentials. This command will perform the following steps:

- Builds all required Docker images.
- Checks if the provided Google credentials are valid by attempting to download an app.
- Checks if an Android device is connected.

The script should print *OK* to the console. Depending on the device resources, this may take up to 20 minutes.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The app downloading and preparation process from the Play Store, as described in Section 5.1, is functional and yields a large dataset for further analysis. This is proven by experiment *E1*.

(C2): TapTrap is effective on Android 15 as described in Section 3 of the paper. This is proven by experiment *E2*.

(C3): A large-scale analysis of 99,705 apps using the animation detection pipeline (cf. Section 5.2) identified 61 unique animations that bypassed the intended duration limit of 3,000ms (Section 5.3.1) and 28 apps containing animations with a maliciousness score of at least 50 (cf. Section 5.3.2). This is proven by experiment *E3*.

(C4): 76.3% of analyzed apps are vulnerable to TapTrap based on the static analysis pipeline provided in Section 6 and Table 3. This is proven by experiment *E4*.

A.4.2 Experiments

(E1): Dataset preparation [15 human-minutes + 1 compute-hour + 20 GB disk]

Preparation: Follow the steps in Section A.3.1 to set up the environment.

Execution: Run `reproducibility/e1.sh <EMAIL> <TOKEN> <OUT>` in the repository and replace `<EMAIL>` with the Google account email, `<TOKEN>` with the AAS token, and `<OUT>` with the desired output directory. The script performs a Play Store crawl and download and prepares the APKs for analysis. Due to time constraints, the crawl stops after 25,000 package names, and only a random subset of 50 apps is attempted for download.

Results: The script should output `OK`.

Alternatively, manually verify that:

- `<OUT>/apps.csv` contains $\geq 15,000$ package names (lower bound after crawling 25,000 package names)
- `<OUT>/apps` contains ≥ 30 apps (lower bound after downloading 50 apps)
- Note that lower bounds are due to regional restrictions and possible device limitations.

(E2): TapTrap functionality [30 human-minutes + 10 compute-minutes + 15 GB disk]

Preparation: Follow the steps in Section A.3.1 to set up the environment, then run `reproducibility/e2.sh` from the repository root to install and launch the app. Note that running an emulator on x86 with nested virtualization is extremely slow and may cause the attack to fail.

Execution: Click the "Start" button in the app to initiate the attack, then click "Click here". The app secretly opens a camera permission prompt and attempts to trick the user into granting access.

Results: The app should display *"Permission granted"* without the user being aware of granting the permission.

Alternatively, manually verify that camera access has been granted to the app: long-press the app icon, select "App info", then "Permissions".

(E3): Detection of potentially malicious animations [15 human-minutes + 45 compute-minutes + 100 GB disk]:

Preparation: Follow the steps in Section A.3.1 to set up the environment, including downloading the APKs.

Execution: Run `reproducibility/e3.sh <APK_DIR> <OUT_DIR>` in the repository root, where `<APK_DIR>` is the input directory containing the APKs and `<OUT_DIR>` is the output directory for results. The script analyzes a random subset of 500 apps, plus a fixed subset of apps that span the 61 animations exceeding 3,000 ms in duration and those that have a score of at least 50.

Results: The script should output `OK`.

Alternatively, manually verify that the generated report at `<OUT_DIR>/report.tex` states:

- `maltapNumberUniqueAnimationsExtendedDuration`: 61 unique animations exceeding the 3,000 ms duration threshold were found (cf. Section 5.3.1).
- `maltapNumberAppsAnimationsScoreMin`: 28 apps containing at least one animation with a maliciousness score of at least 50 were found (cf. Section 5.3.2).

(E4): Detection of vulnerable apps [15 human-minutes + 4 compute-hour + 100 GB disk]

Preparation: Follow the steps in Section A.3.1 to set up the environment, including downloading the APKs.

Execution: Run `reproducibility/e4.sh <APK_DIR> <OUT_DIR>`, where `<APK_DIR>` is the directory containing the APKs and `<OUT_DIR>` is the output directory for results. The script executes the vulnerable app detection pipeline on the app dataset.

Results: The script should output `OK`.

Alternatively, manually verify that 76.3% ±10% of analyzed apps are vulnerable by inspecting the `vulntapAmountAppsMinOneActivity VulnerablePercent` macro in the generated report located at (`<OUT_DIR>/report.tex`). The error margin accounts for possible variability introduced by analyzing only a subset of apps.

A.5 Notes on Reusability

To foster future research and make it easier for others to build on our work, we provide detailed documentation in the `README.md` files included in each subdirectory. These files include troubleshooting information, describe how to adjust the analysis pipeline (e.g., changing the level of parallelism), explain the code organization and module structure, and outline usage outside Docker environments.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.